Cloud Computing

Virtual Resources and Distributed Cloud Applications

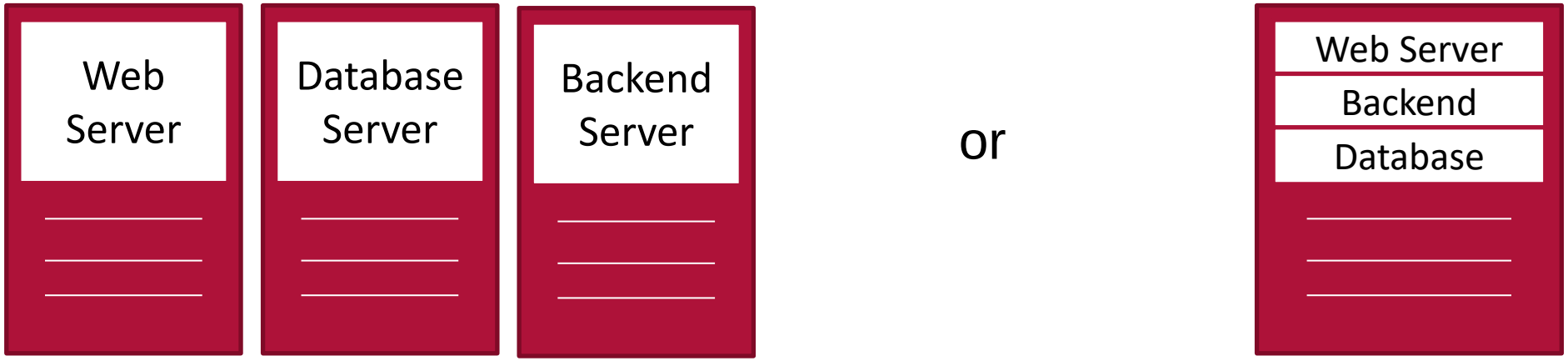
Chapter 3: Management of Virtual Resources

Distributed and Operating Systems
Electrical Engineering and Computer Science
Technische Universität Berlin

Overview

- Intro
- Cloud Operating Systems
  - OpenStack
- Infrastructure-as-Code
  - Ansible
- Container Orchestration
  - Kubernetes
- DevOps, Continuous Integration, and Continuous Delivery
  - Spinnaker, GitLab

“Iron Age”: Bare-Metal Servers



- Running components of an application on one or multiple physical servers:
  - Costs and agility: Purchase, housing, maintenance
  - Fault tolerance: Servers are single points of failure
  - Resource utilization: Servers are often underutilized, but sometimes also bottlenecks (i.e. with dynamic load)

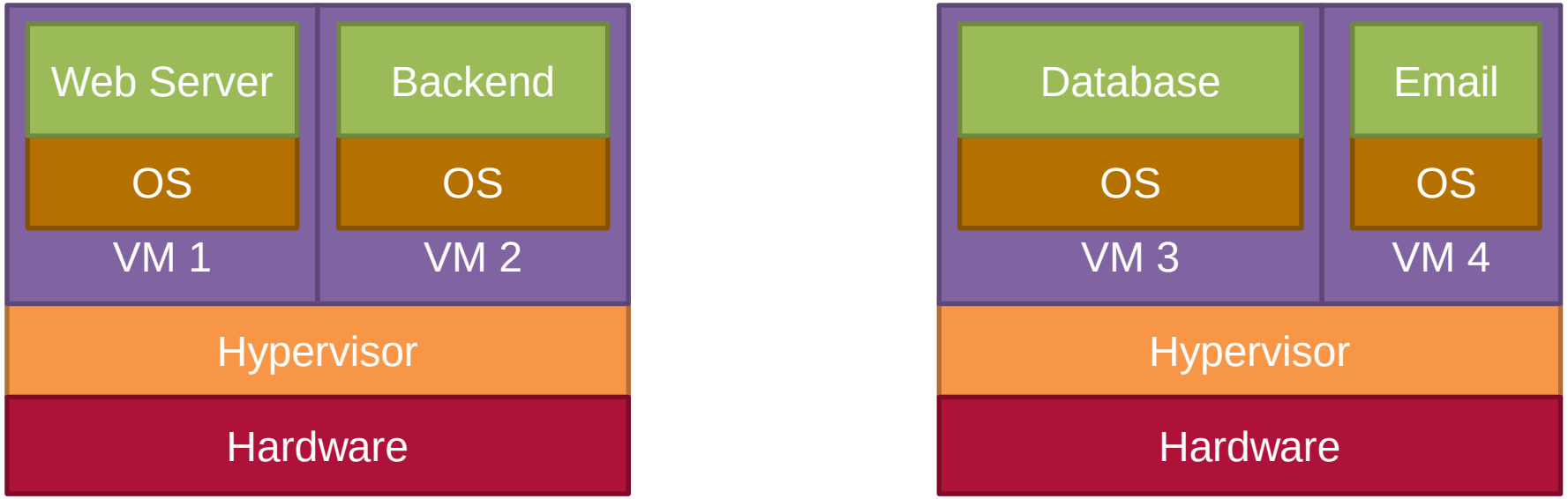
Bare-Metal Servers in one Organization

- Server consolidation: Manually assign components from different applications to a set of physical nodes

- Only possible within one organization without isolation
- Static assignment, while loads may vary dynamically

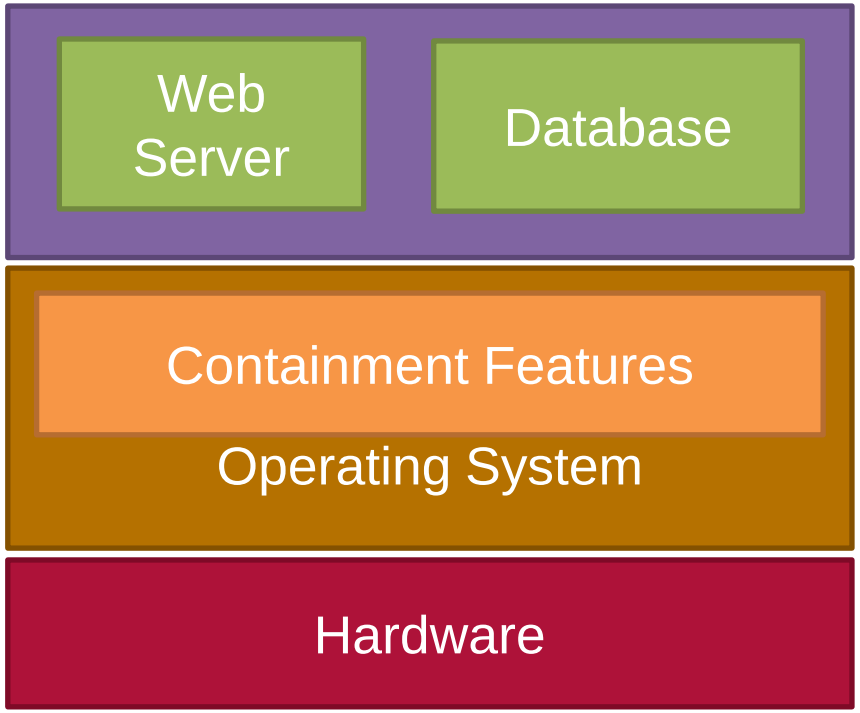
Recap: Virtual Machines

- System virtualization: Same architecture, but possibly different OS and resource configuration (#cores, RAM, storage) and isolation of applications
- Same physical host can run multiple virtual machines
- Live migration of VMs and snapshots for load balancing and fault tolerance



Recap: Containers

- Motivation: VM images are large (contain OS) and starting VMs takes longer than processes (10x)
- Containers: Same OS kernel, but different resource configuration and isolation of contained processes
- Containment using Linux kernel features such as chroot, cgroups, and namespaces



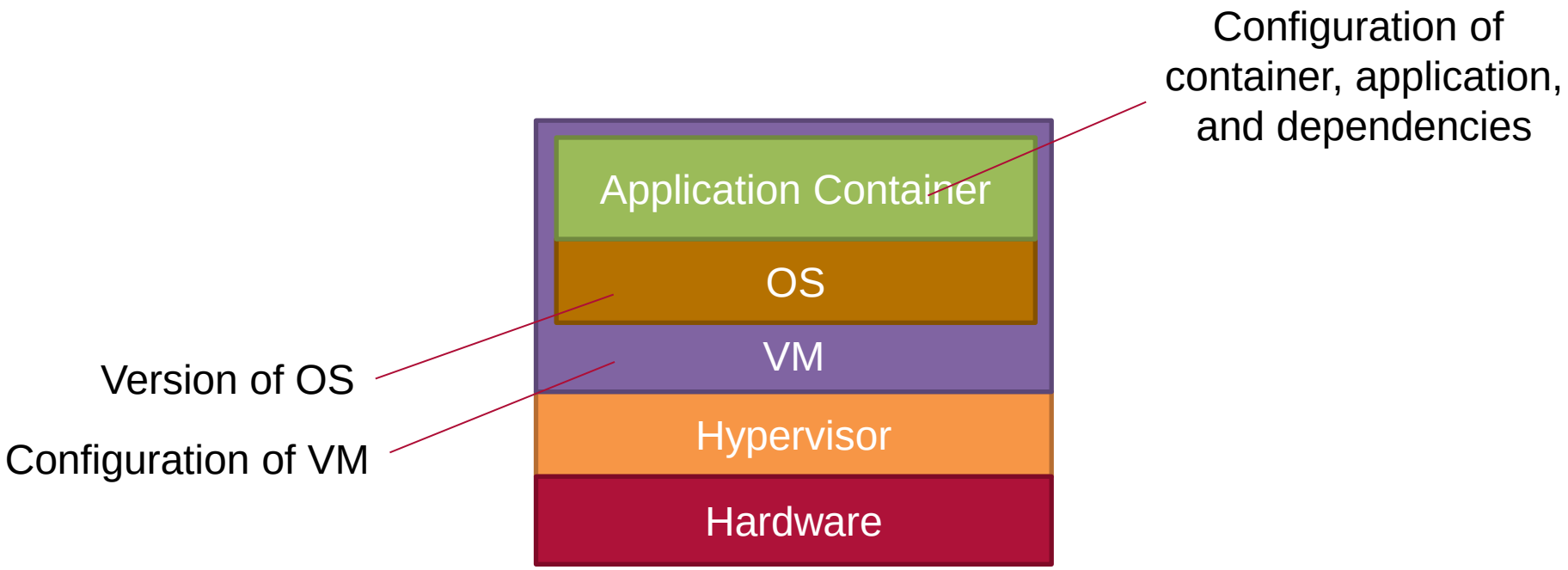
Management of Resources (1/2)

- Now we can create VMs and containers, but management of large sets of resources is still difficult:
  - How to provision VMs and containers?
  - How to configure systems?
  - How to monitor systems and handle failures?
  - How to replicate components and balance loads?

- Cloud Operating Systems: manage large sets of virtual resources running on large sets of physical resources

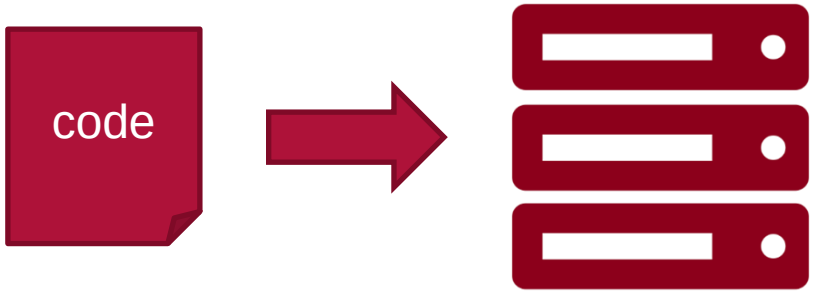
Management of Resources (2/2)

- Major remaining problems in the “Cloud Age”: server sprawl, configuration drift, snowflake servers… → technical debt!



Infrastructure-as-Code

- Define servers, networking, and other infrastructure elements in source code:
  - Configuration of environments
  - Dependencies with specific versions



- Automation tools built around infrastructure definitions ➔ easily rebuild servers (“immutable infrastructure” instead of updating running servers)
- Versioning and sharing of infrastructure definitions

Overview

- Intro
- **Cloud Operating Systems**
  - OpenStack
- Infrastructure-as-Code
  - Ansible
- Container Orchestration
  - Kubernetes
- DevOps, Continuous Integration, and Continuous Delivery
  - Spinnaker, GitLab

Cloud Operating Systems (1/3)

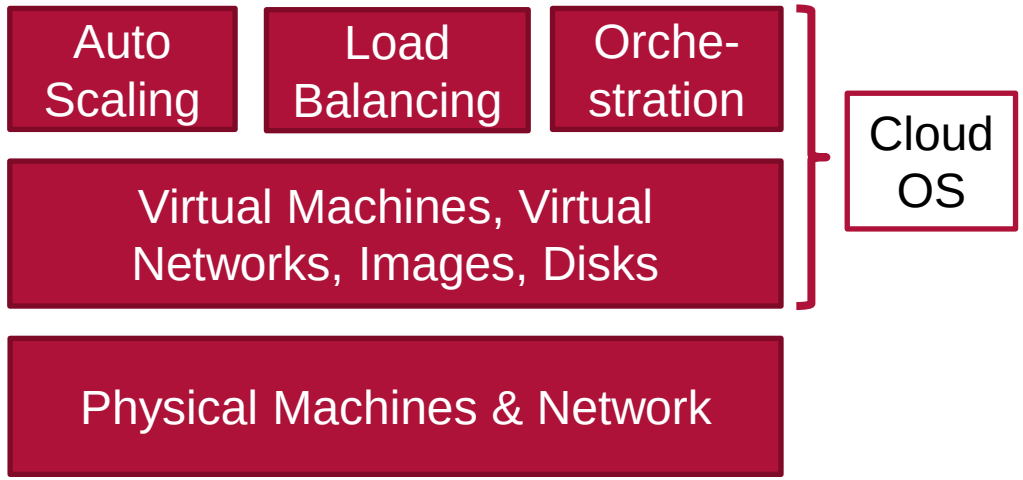
- Virtualization is immensely useful, but managing many virtual machines manually is impractical
- Therefore: Cloud Operating Systems
  - Control large pools of compute, storage, and networking resources
  - Provides dashboards and APIs for datacenter operators (administration) and users (provisioning) that abstracts infrastructures (e.g. the different hypervisors)

- Open-source systems: OpenStack, OpenNebula
- Commercial public clouds: e.g. EC2, GCE, Azure, Alibaba Cloud, Digital Ocean

Cloud Operating Systems (2/3)

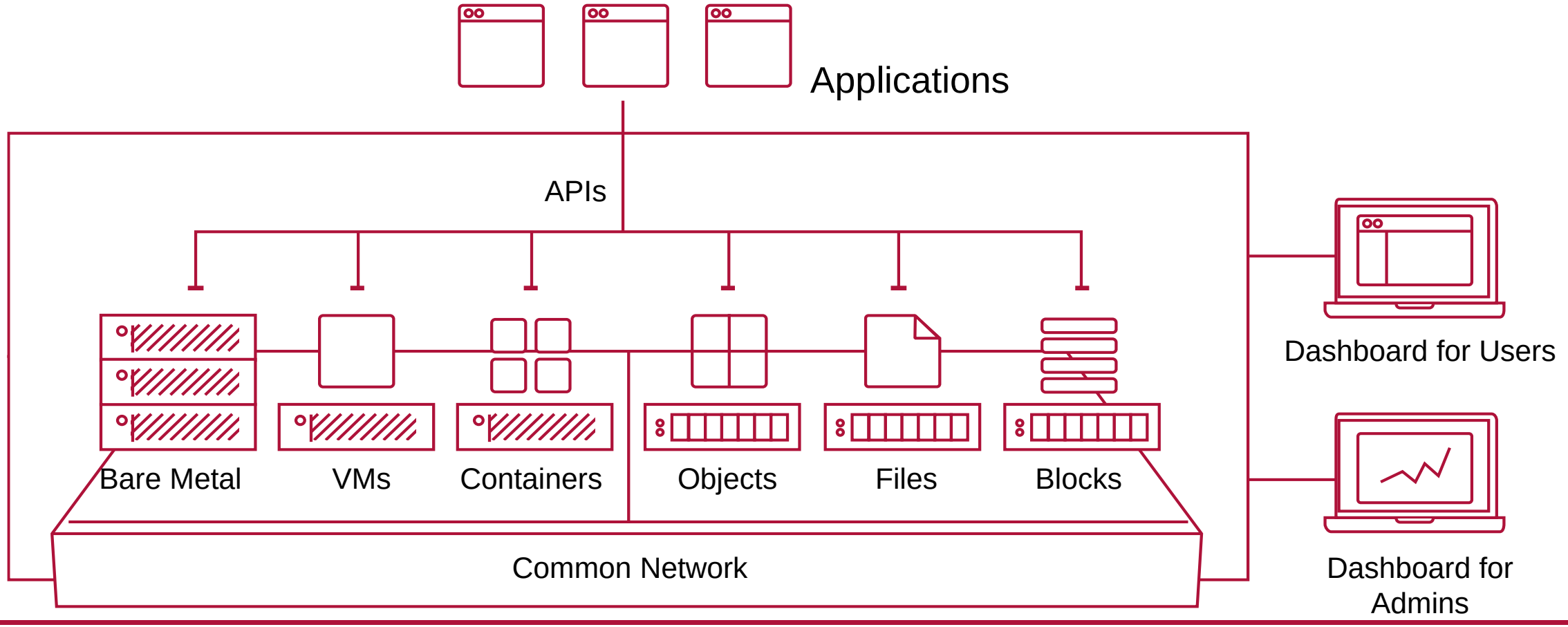
- Basic functions: User Interface and APIs for
  - spawning and maintaining VMs,
  - virtual networking,
  - managing images and virtual disks

- Advanced functions:
  - auto scaling,
  - load balancing,
  - and orchestration



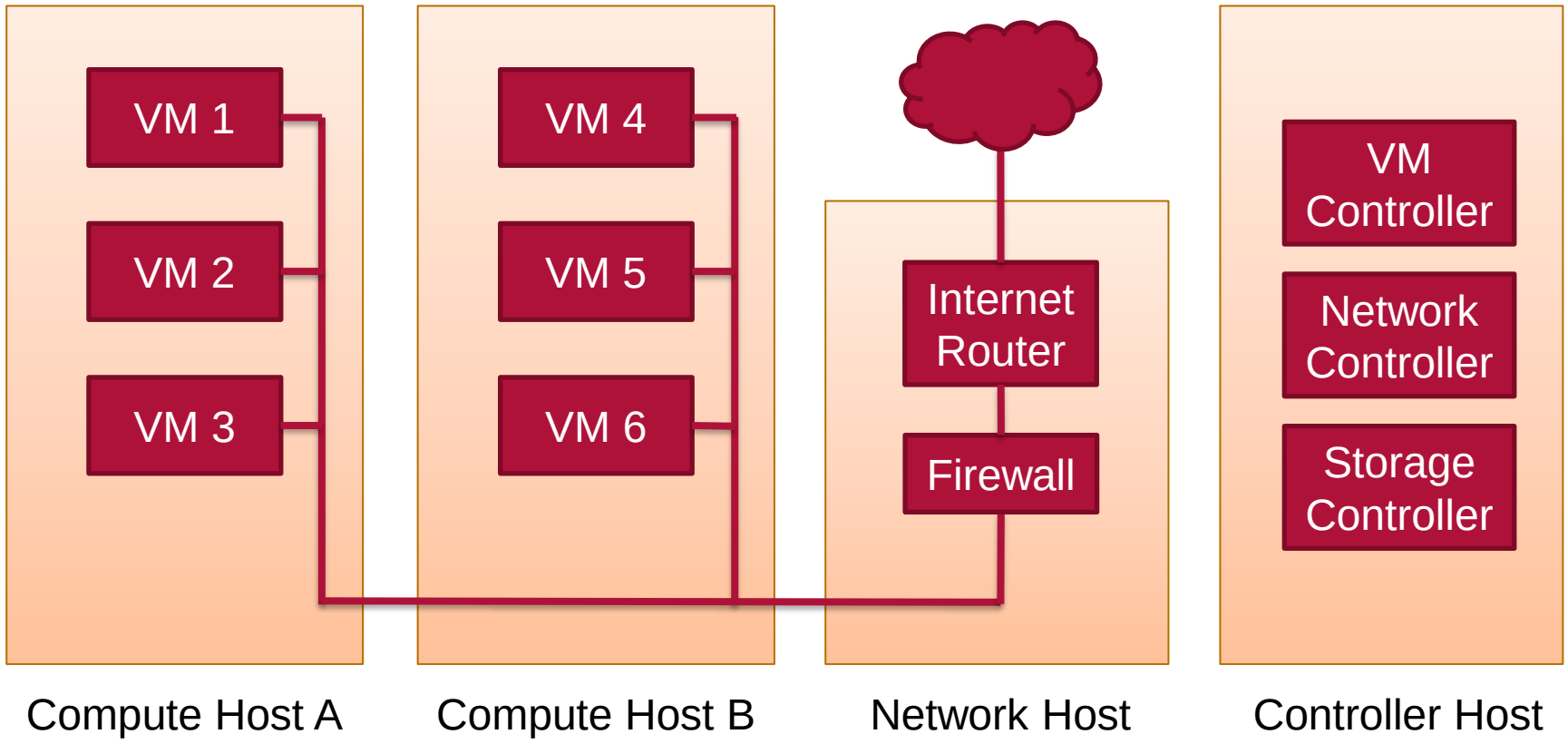
Cloud Operating Systems (3/3)

- Interfaces to manage and provision virtual resources



Managed Cloud Environment

- Different roles for physical hosts: compute hosts, storage hosts, network hosts, and controller host

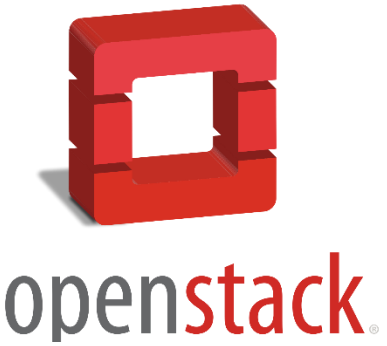


Overview

- Intro
- Cloud Operation Systems
  - **OpenStack**
- Infrastructure-as-Code
  - Ansible
- Container Orchestration
  - Kubernetes
- DevOps, Continuous Integration, and Continuous Delivery
  - Spinnaker, GitLab

OpenStack

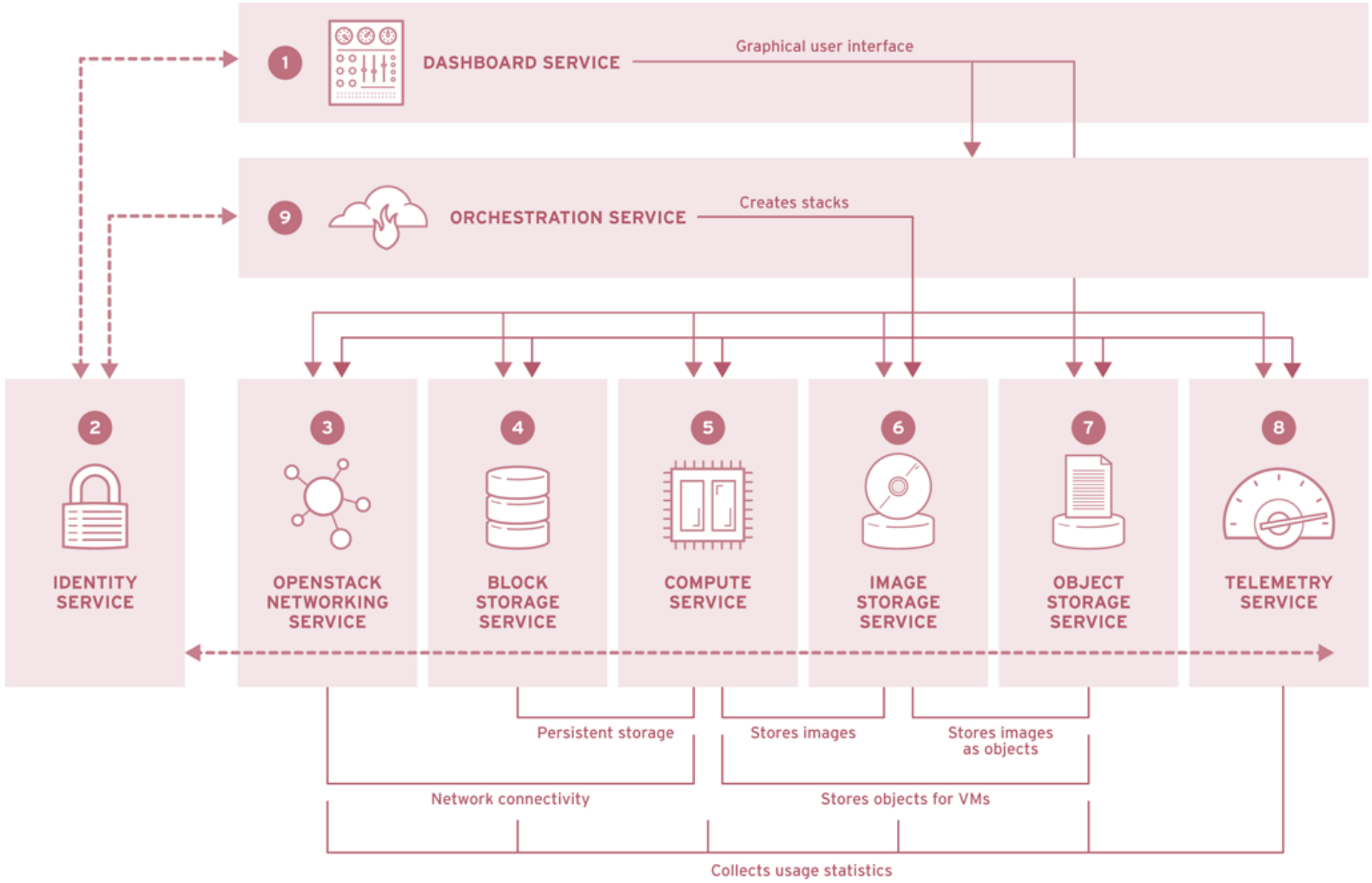
- Open-source cloud operating system
- Mostly deployed as IaaS, providing virtual machines and other resources to users



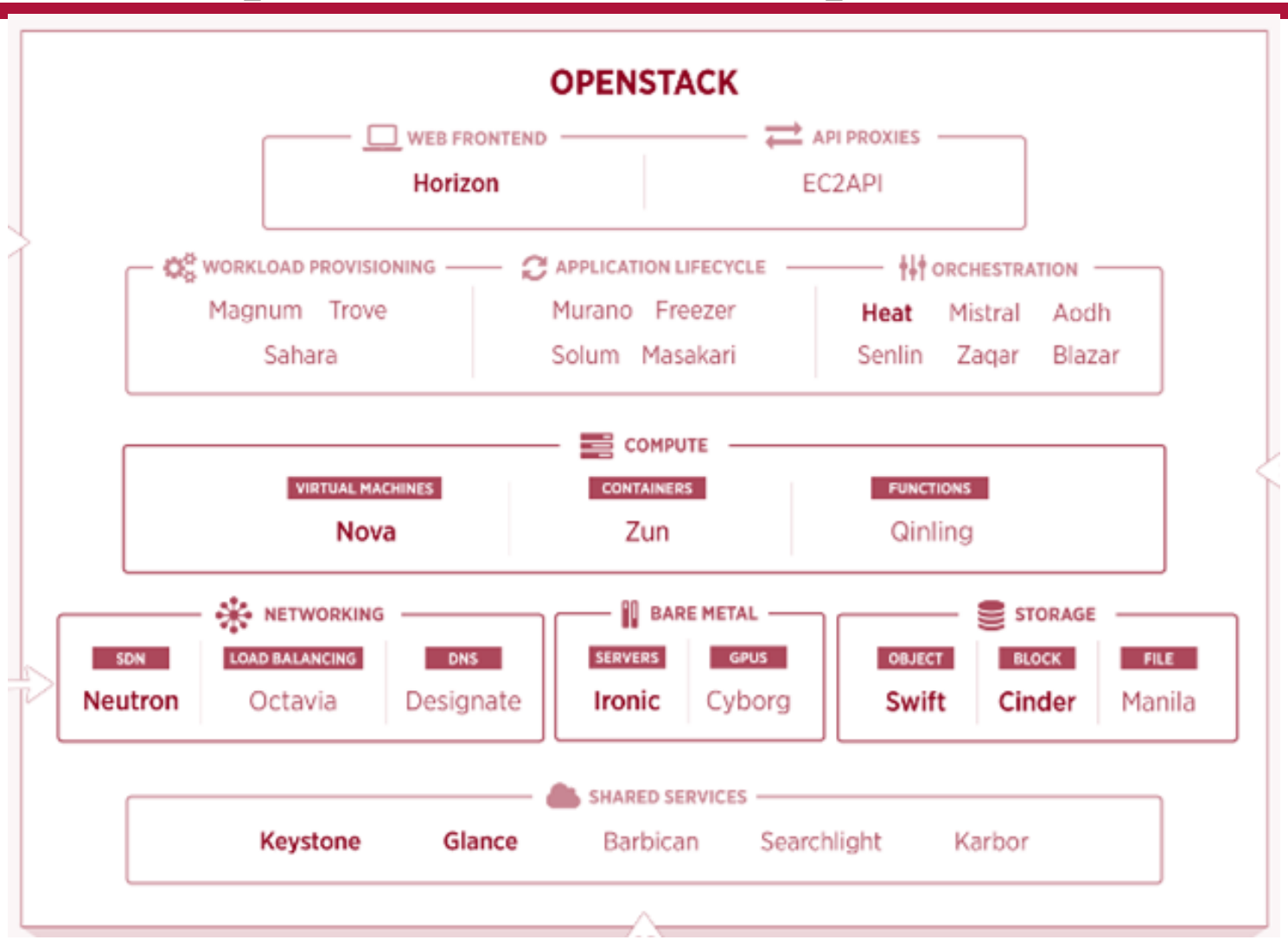
- Started in 2010 by Rackspace Hosting and NASA
- Contributions from IBM, Red Hat, HP, Cisco, Google, Oracle, EMC, VMware

- Most popular open cloud platform in both industry and academia

Core Components of OpenStack (1/2)



Core Components of OpenStack (2/2)



Compute Service: *Nova*

- Allows to create and manage virtual machines on demand from images
- Schedules virtual machines to run on physical nodes
- Defines drivers that interact with underlying virtualization mechanisms

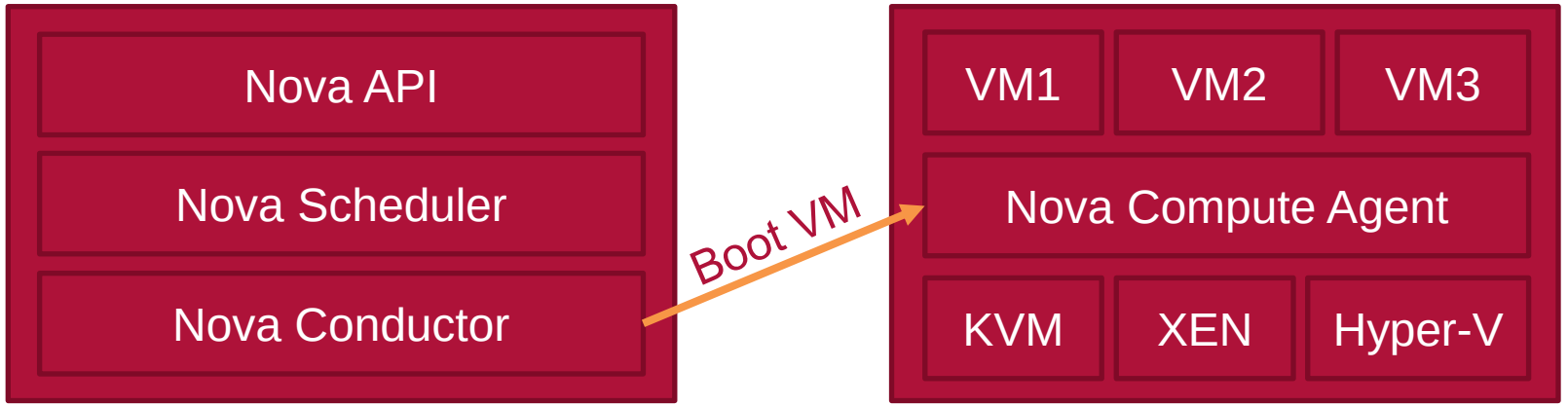
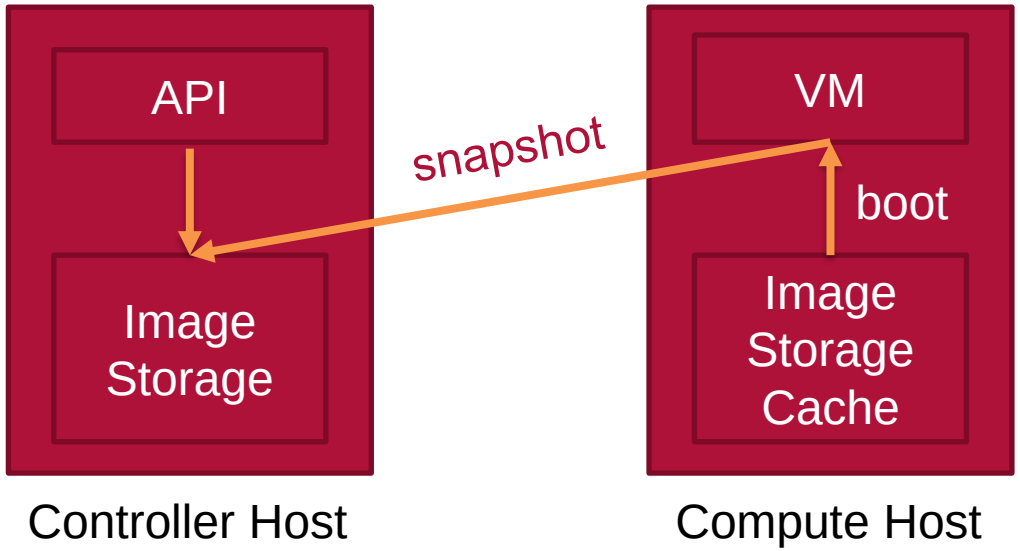


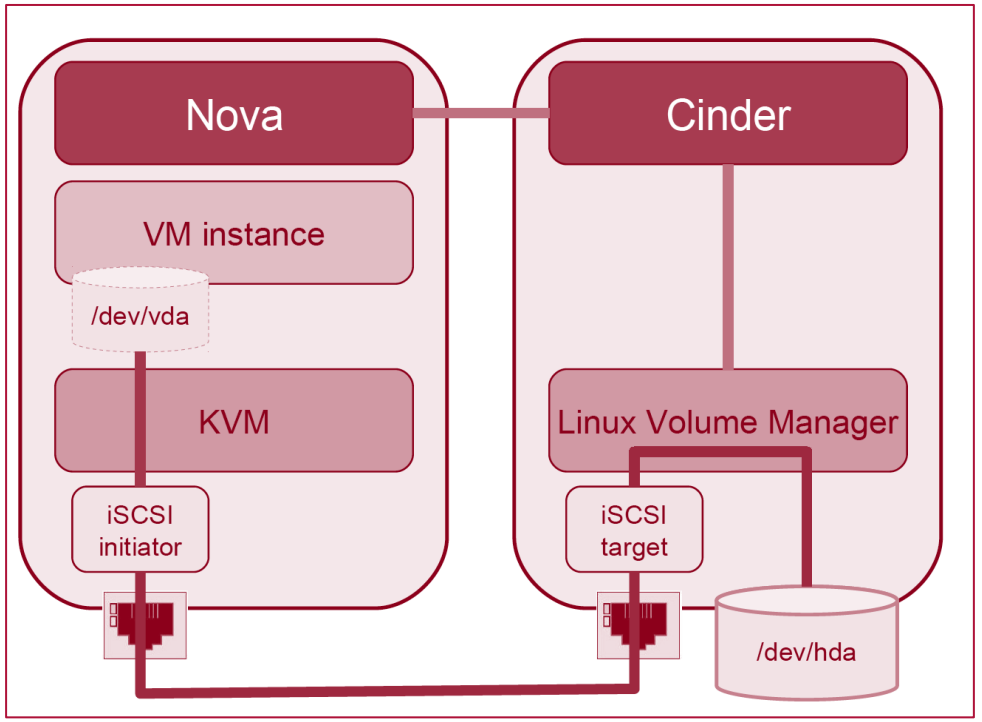
Image Service: *Glance*

- Registry for virtual disk images
- Users can add new images or take snapshots of existing VMs
- Snapshots can be used as templates for new servers



Block Storage Service: Cinder

- Provides network-backed virtual block storage volumes
- Enables live migration of VMs and supports replication
- Multiple storage backends available

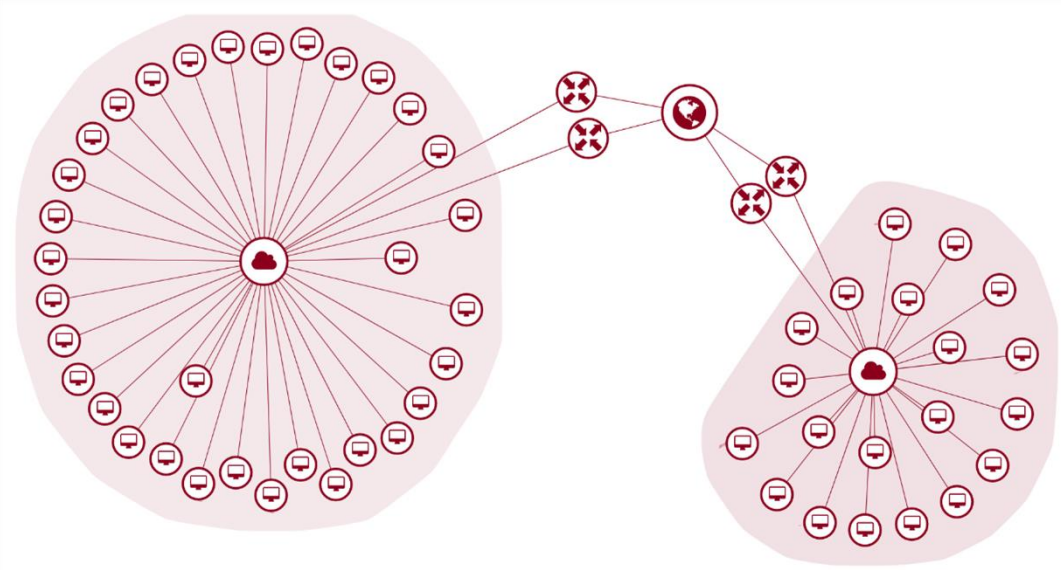


Virtual Networking: *Neutron*

- Provides API for networking between virtual machines and physical network

- Interconnects virtual machines
  - within one hypervisor
  - between several hypervisors
  - with the Internet

- Provides additional network functions
  - Like firewalls, VPN, public IPs …

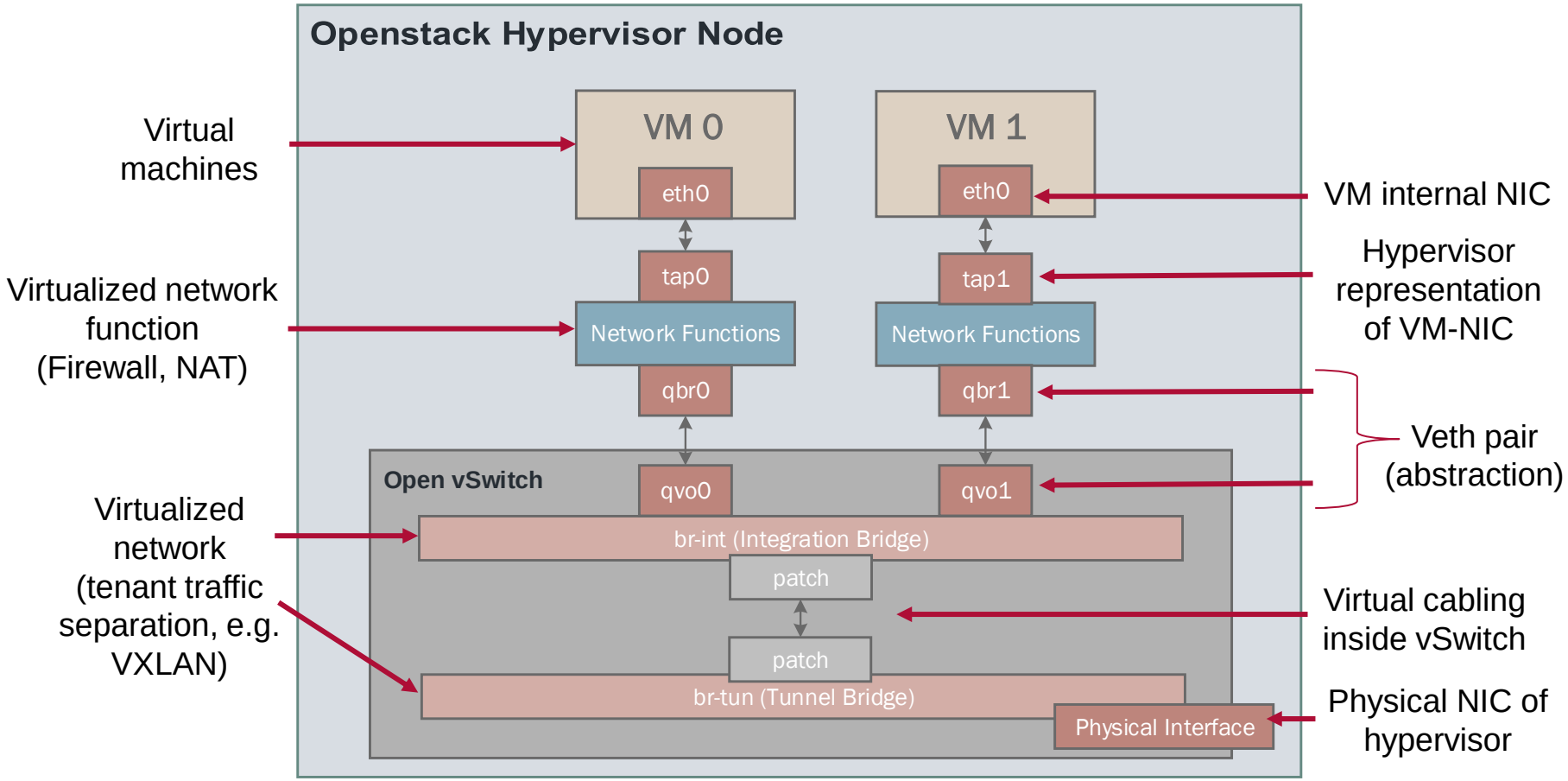


Virtual Switches & SDN

- Virtual switches: Virtual switching stack that enables network automation through programmatic extensions and standard protocols (e.g. OpenFlow)

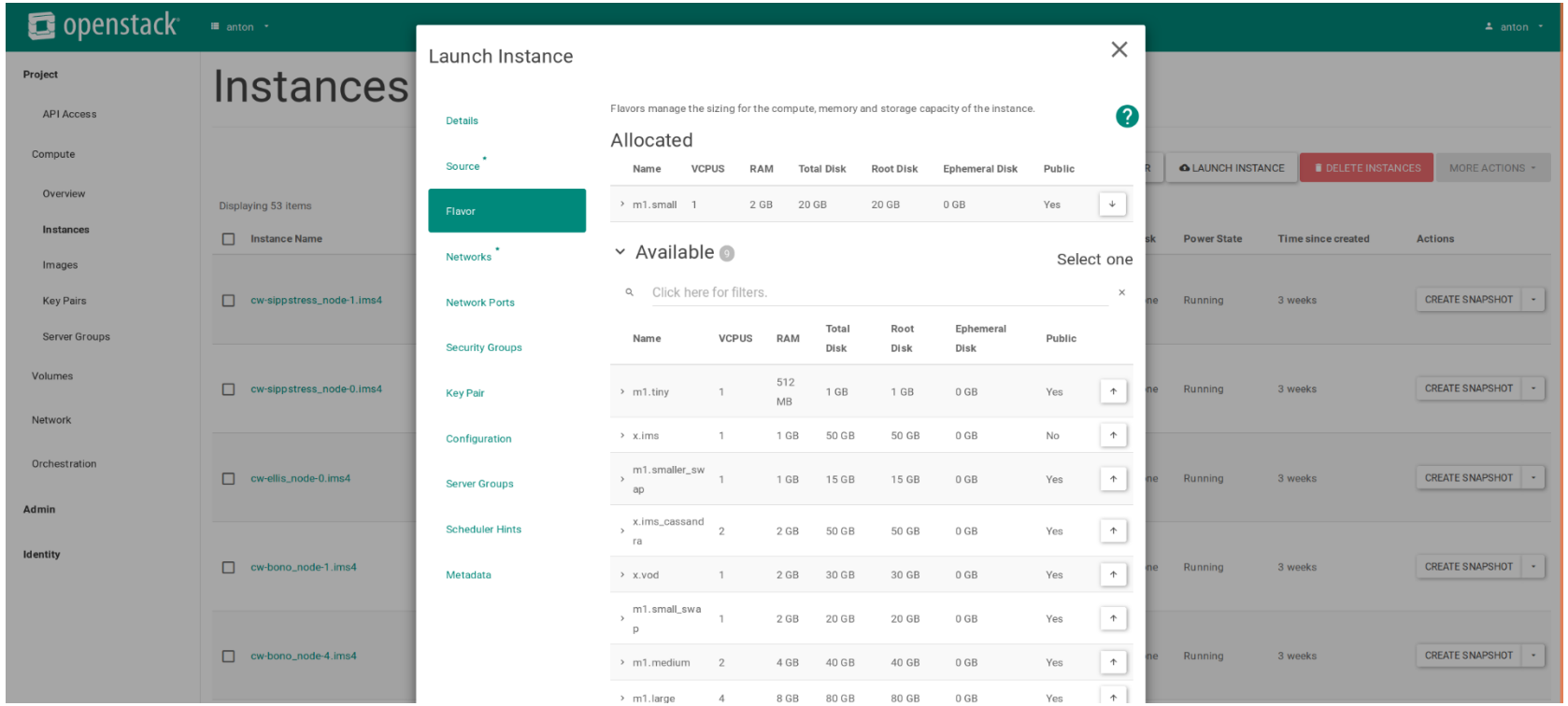
- Software-Defined Networking (SDN)
  - Goal: Simplify network administration
  - Separation of control plane from data plane
  - Allows on the fly configuration of switches (controller-based decision making)

Virtual Networking in OpenStack



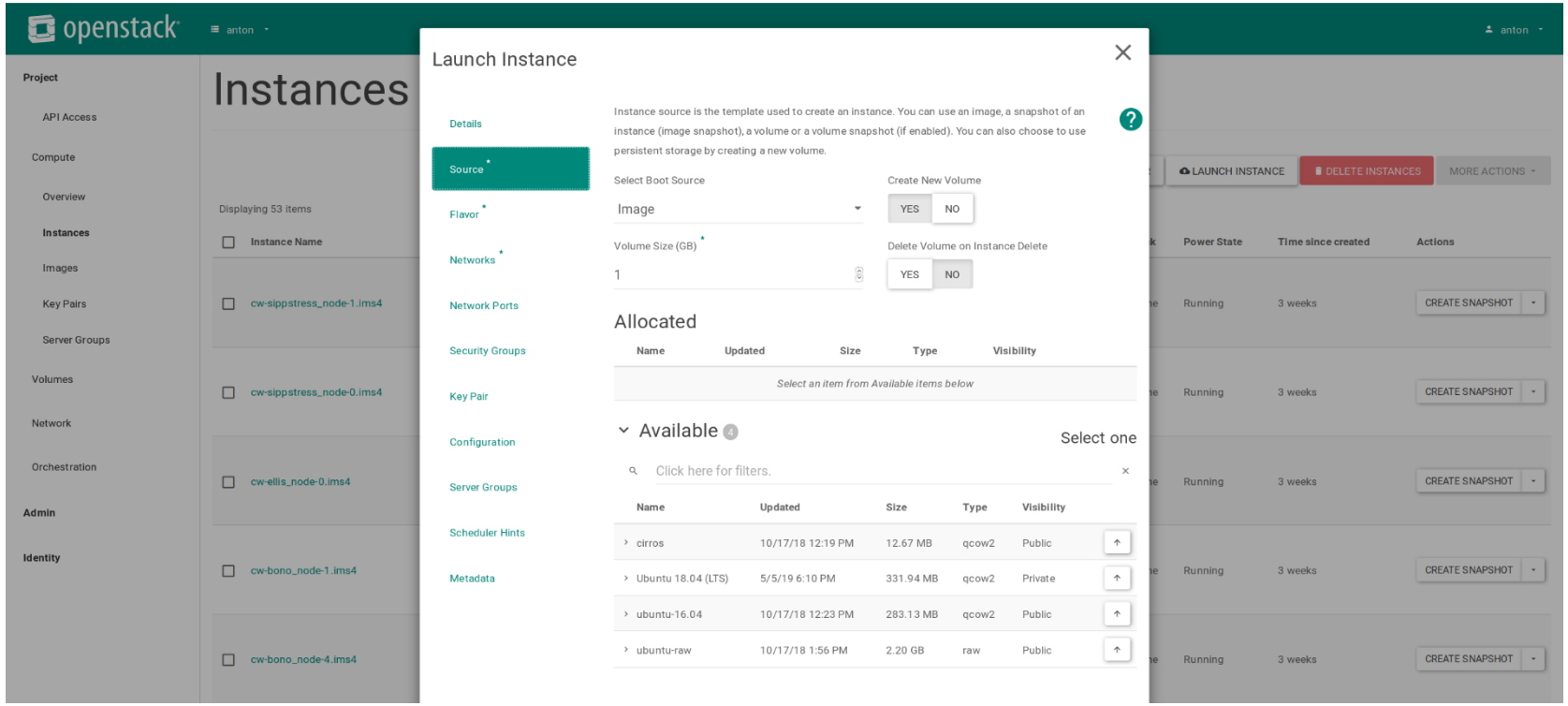
OpenStack Walkthrough 1,2: Virtual Resources with *Nova*

- Users authenticate first with *Keystone* (identity management)
- Users request VMs from *Nova* (compute), regardless of underlying resources and hypervisors



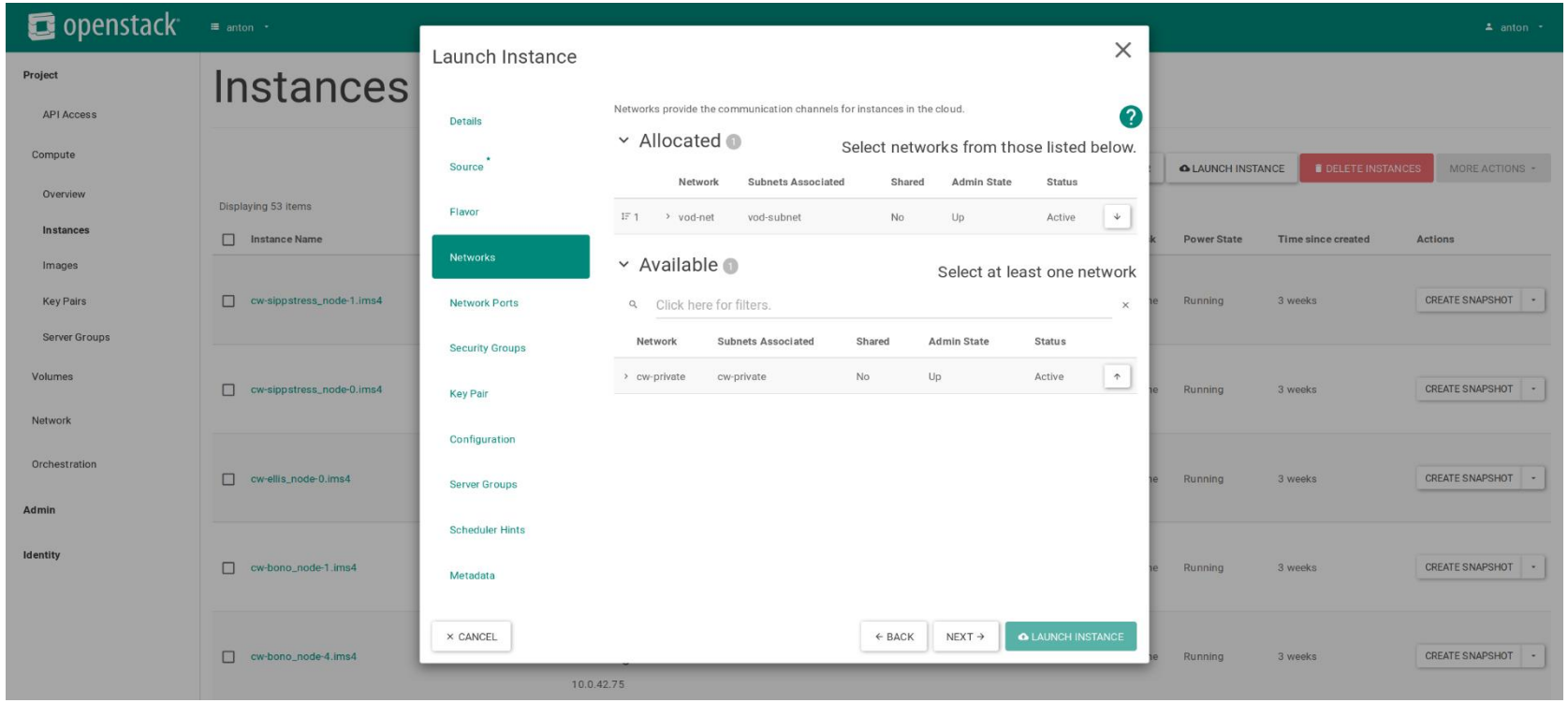
OpenStack Walkthrough 3: Images via *Glance*

- Users can select an OS image for their VMs from *Glance* (Image Service)



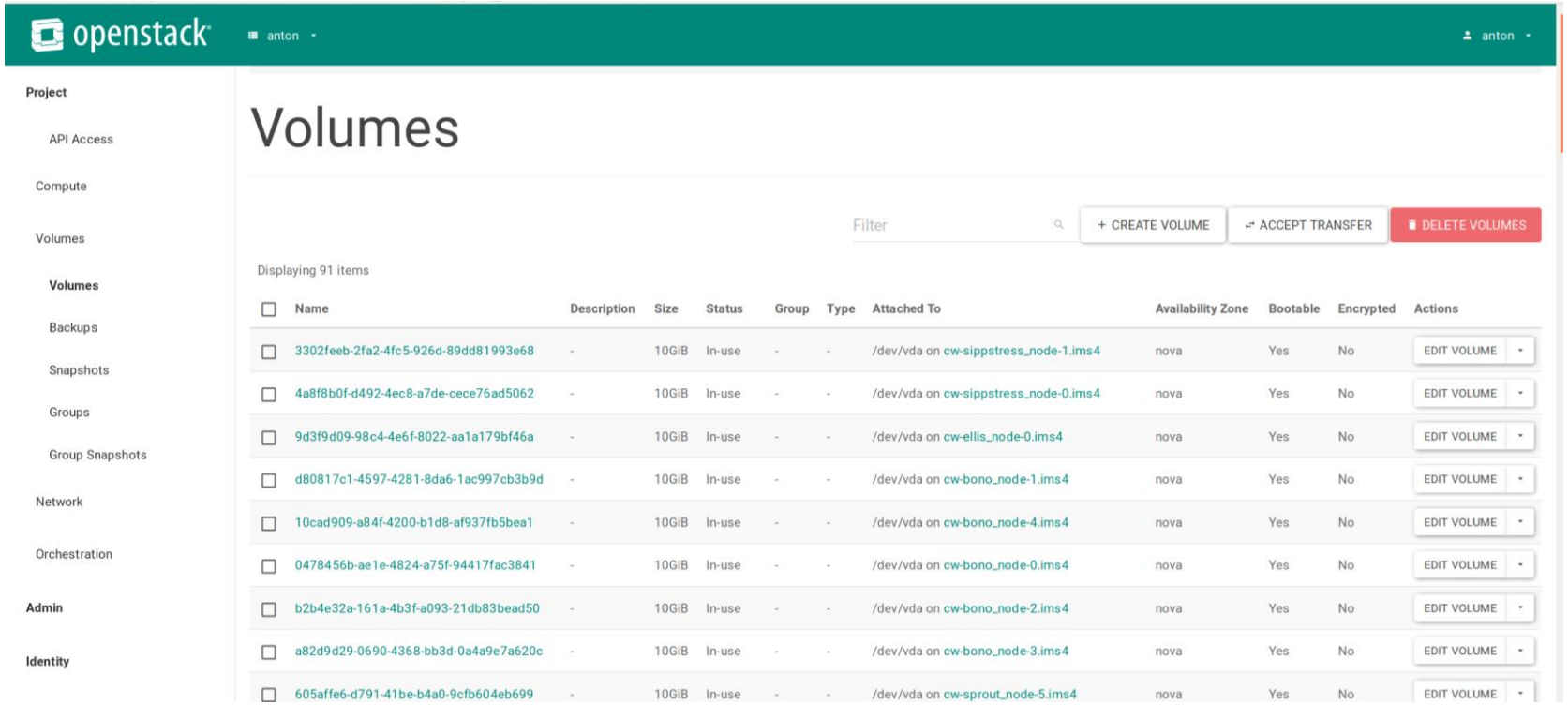
OpenStack Walkthrough 4: Network Resources via *Neutron*

- Users can provision network resources for their VMs from *Neutron* (Virtual Networking), e.g. firewalls and public IPs



OpenStack Walkthrough 5: Storage via *Cinder*

- Users can provision additional storage for their VMs via *Cinder* (Storage Service)



Overview

- Intro
- Cloud Operating Systems
  - OpenStack
- **Infrastructure-as-Code**
  - Ansible
- Container Orchestration
  - Kubernetes
- DevOps, Continuous Integration, and Continuous Delivery
  - Spinnaker, GitLab

Motivation

- Manual configuration of servers usually leads to
  - Configuration Drift: configuration of servers changed manually over time (without documentation)
  - Snowflake Servers:
    - Have a unique configuration
    - Are not reproducible when hardware dies
    - Are difficult to mirror for a test environment
    - Deliberate vs. default configuration
    - Host-specific vs. general configuration

Open Questions

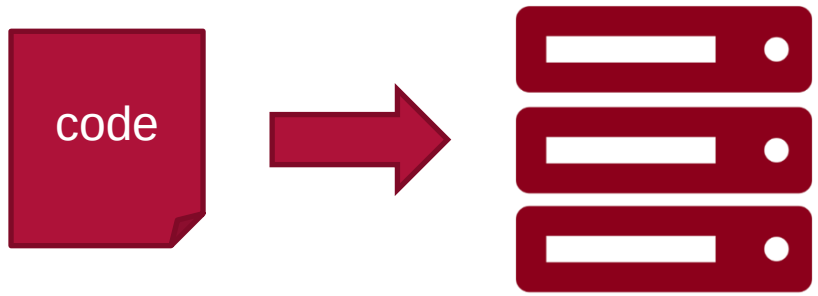
- What if a server fails?
- How to mirror a production environment for testing?
- How to keep a cluster of servers consistent?

- Disk snapshots allow to recover and mirror servers, but
  - still not straightforward to understand the state of servers
  - do not allow managing multiple servers that are mostly similar, but specific in a few configurations

Infrastructure as Code

- Goal: Make systems easily reproducible, support testing, and configurations more understandable

- Solution: One single source of truth with a clear declaration of host-specific configuration → infrastructure definition files
  - that can be executed
  - that can be versioned
  - that can be shared



- Only use these definitions and automation tools, without manually updating single servers

Overview

- Intro
- Cloud Operating Systems
  - OpenStack
- Infrastructure-as-Code
  - **Ansible**
- Container Orchestration
  - Kubernetes
- DevOps, Continuous Integration, and Continuous Delivery
  - Spinnaker, GitLab

Ansible

- Automation language and engine

- Configuration management
  - i.e. configure existing servers
- Resource orchestration
  - i.e. allocate and provision new servers

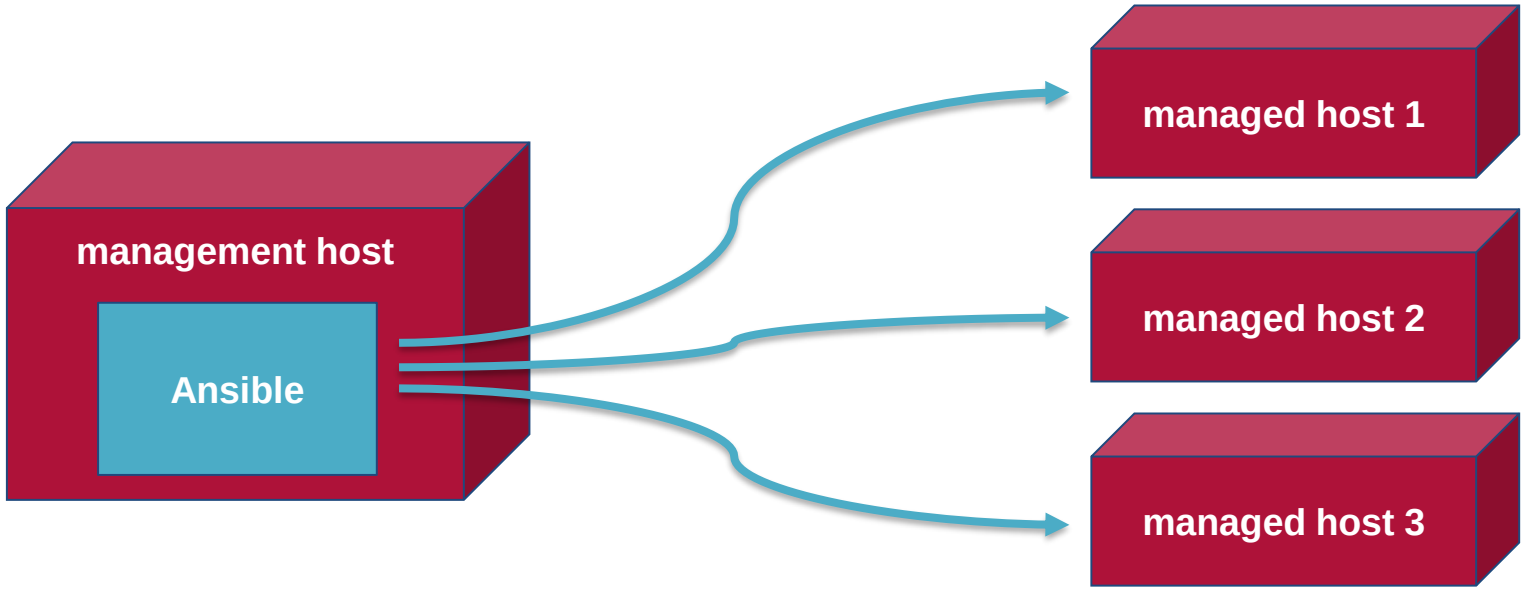
- Started in 2012, acquired by Red Hat in October 2015

- Well-documented conventions and best practices, but no artificial limits, so users can choose what to ignore

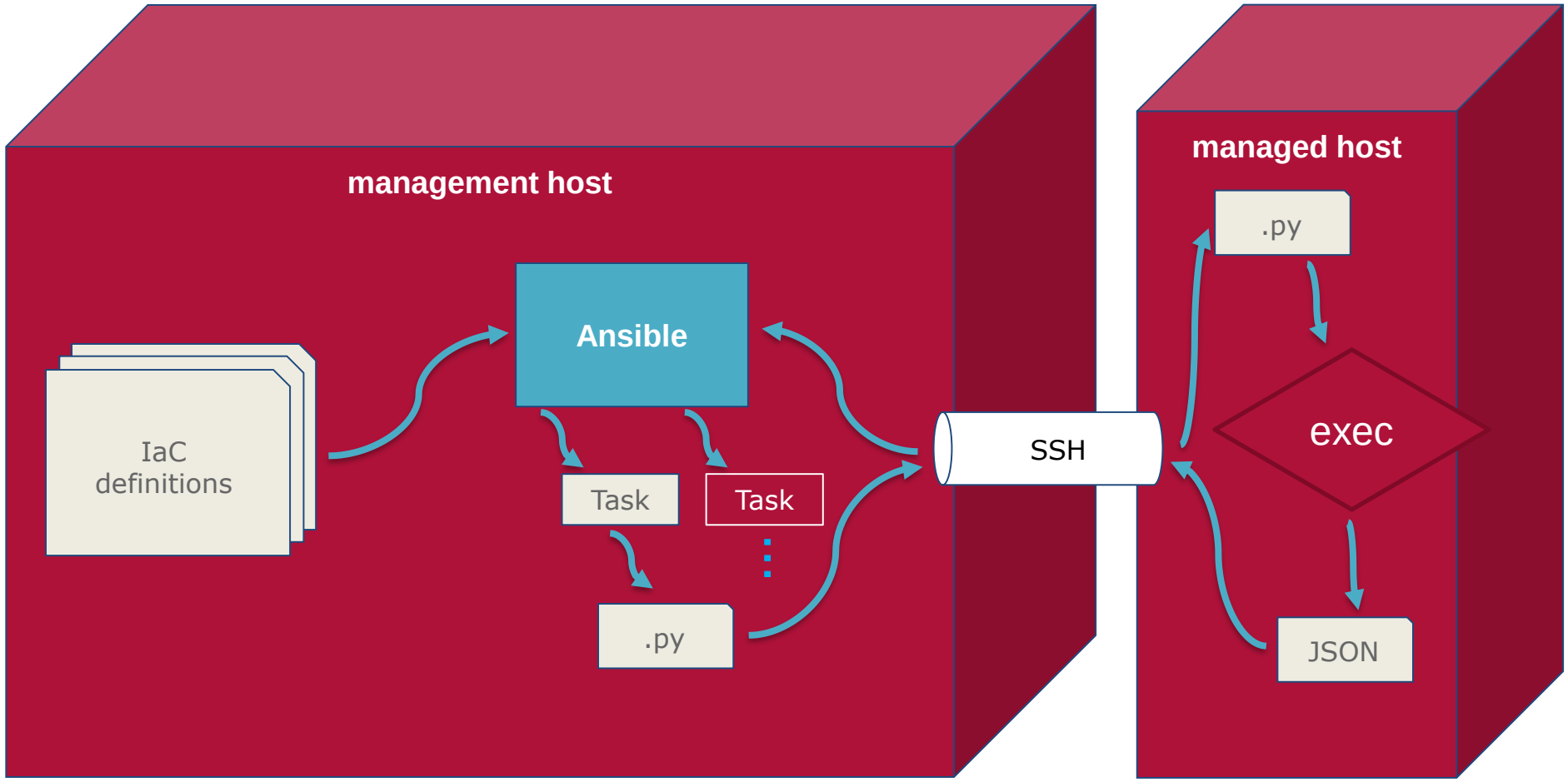
Ansible

- Multi-platform
  - Operating systems: Linux, Unix-like, MacOS
  - Orchestration-related tools: LXC, libvirt, VMware, …
  - IaaS services: AWS, Azure, OpenStack, …
- Well-established technologies as foundation
  - Python
  - SSH (PowerShell remoting, respectively)
  - YAML + Jinja templating
- Design: Declarative and agent-less

Agent-Less Architecture (1/2)



Agent-Less Architecture (2/2)



Ansible Tasks (1/2)

- Declarative i.e., define the desired state
- Opposed to imperative i.e., define how to achieve the desired state
- Implies idempotency i.e., applying the configuration again leads to same state
  - # that's the intention at least; you can, of course, create non-idempotent tasks

```
# an example "task":
- name: "enforce permissions for .ssh directory of {{ ansible_user_id }}"
  # ↑ "name" is optional but strongly encouraged
  file: # ← the module name, ↓ followed by module parameters
    dest: "{{ ansible_user_dir }}/.ssh"
    state: directory
    mode: u+rX,go-rwx
    recurse: yes
```

Ansible Tasks (2/2)

- But to which hosts should this task be applied to?

```
# an example "task":
- name: "enforce permissions for .ssh directory of {{ ansible_user_id }}"
  # ↑ "name" is optional but strongly encouraged
  file: # ← the module name, ↓ followed by module parameters
    dest: "{{ ansible_user_dir }}/.ssh"
    state: directory
    mode: u+rX,go-rwx
    recurse: yes
```

Ansible Playbooks (1/2)

- Ansible's execution unit: Playbooks map everything to hosts
  - Tasks
  - Roles
  - Variables
- Further features include:
  - Rolling updates
    - Configure only n hosts at a time
    - Configure only x% of host at a time
  - Tolerate percentage of hosts where configuration fails
  - …

```
# file: my_playbook.yml

# a "Play":
- name: configure internal Web servers
  hosts: webservers
  roles: webserver
  tasks:
    - name: forbid external connections
      iptables:
        chain: INPUT
        source: !192.168.1.1/24
        jump: DROP
```

Ansible Playbooks (2/2)

- But how does Ansible know the hosts "webservers"?

```
# file: my_playbook.yml

# a “Play”:
- name: configure internal Web servers
  hosts: webservers
  roles:  webserver
  tasks:
    - name: forbid external connections
      iptables:
        chain: INPUT
        source: !192.168.1.1/24
        jump: DROP
```

Ansible Inventory

```
# no FQDNs required, as long as
# ``ssh <hostname>`` works (~/.ssh/config)
mail
[webservers]
w1.example.com
w2.example.com
[dbservers]
d1.example.com
d2.example.com
[appservers]
# we can override variables:
a01.example.com www_user=web
# we can use a pattern-like syntax:
a[02-12].example.com

# we can define variables for groups:
[appservers:vars]
www_user=www-data

# a group of groups
[rootservers:children]
appservers
dbservers
```

```
# alternatively, we can use YAML:

all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        w1.example.com:
        w2.example.com:
    dbservers:
      hosts:
        d1.example.com:
        d2.example.com:
  ...
```

- Inventories can also be obtained from external scripts / executables (e.g. IaaS clouds)

Ansible Variables (1/2)

- Variables can be defined in many places
  - Playbooks, Tasks, Inventory, CLI args, roles, facts, …
  - But scattering variables all over the place is discouraged, of course
  - e.g., using role, group and host variables can suffice
- "facts"
  - A heap of auto-detected variables: e.g., Uptime, chroot?, user home, SSH host key, Python version, total and free memory, host name, FQDN, environment variables, block devices, mounts, interfaces, BIOS version, processor specs, OS name and version, …
- Jinja templating also in variables files
  - e.g., app_user: "{{ ansible_env.SUDO_USER|default(ansible_user_id) }}"

Ansible Variables (2/2)

- Host and group variables
  - e.g., in project directory:
  - # e.g., set inventory to ./inventory.ini:

```
.ansible.cfg
inventory.ini
my_playbook.yml
group_vars/
  # applies to all hosts:
  all.yml
  # applies to webservers:
  webservers.yml
host_vars/mail.yml
```

```
# no FQDNs required, as long as
# ``ssh <hostname>`` works
(~/.ssh/config)
mail
[webservers]
w1.example.com
w2.example.com
[dbservers]
d1.example.com
d2.example.com
[appservers]
# we can override variables:
a01.example.com www_user=web
# we can use a pattern-like syntax:
a[02-12].example.com
# we can define variables for groups:
[appservers:vars]
www_user=www-data
# a group of groups
[rootservers:children]
appservers
dbservers
```

Ansible Roles (1/2)

- Roles
  - e.g., in project directory:

```
roles/webserver
  # default variables:
  defaults/main.yml
  # tasks to be executed:
  tasks/
    main.yml
    # ↑ might include:
    certbot.yml
  …
```

Ansible Roles (2/2)

- Roles
  - e.g., in project directory:

```
roles/webserver
  ...
  # e.g., restart service on
  # configuration changes:
  handlers/main.yml
  # e.g., declare dependencies to
  # other roles:
  meta/main.yml
```

Summary of IaC in Ansible

- Inventory
  - List of managed hosts
  - Groups of hosts (and groups of groups)
- Roles
  - Concept to reuse tasks and variables for similar hosts
  - Usually contain (default) variables
- Host and group variables
  - This is where variables should go
- Playbooks
  - "map" roles, tasks, variables to hosts from inventory

Run a Playbook

$ cat my_playbook.yml

```
- name: test SSH connection
  hosts: pim:static
  tasks:
    - name: test if logging into host and running a module remotely works
      ping: # does not require any arguments
```

$ ansible-playbook my_playbook.yml

```
PLAY [test SSH connection] ****************************************************
TASK [Gathering Facts] ********************************************************
ok: [static]
ok: [pim]

TASK [test if logging in and running a module remotely works] ***************
ok: [pim]
ok: [static]

PLAY RECAP ********************************************************************
pim                        : ok=2    changed=0    unreachable=0    failed=0
static                     : ok=2    changed=0    unreachable=0    failed=0
```

Run Adhoc Commands

$ ansible vs2_and_containers -m file -a "path=/run/log/mylogs state=directory"

```
vs2 | FAILED! => {
    "changed": false,
    "msg": "There was an issue creating …: [Errno 13] Permission denied: '/run/log/mylogs'",
    "path": "/run/log/mylogs",
    "state": "absent"
}
static | CHANGED => {
    "changed": true,
    "gid": 0,
    "group": "root",
    "mode": "0700",
    "owner": "root",
    …
}
rproxy | SUCCESS => {
    "changed": false,
    …
}
…
lists | UNREACHABLE! => {
    "changed": false,
    "msg": "SSH Error: data could not be sent …. Make sure this host can be reached over ssh",
    "unreachable": true
}
```

"But my task is dead-simple …

… using Ansible would be overkill."

- How does your shell script handle …
  - … all the return codes?
  - … whitespaces in variables and undefined variables?
    - e.g., `$ rm -rf /$x`
  - … being re-run – do we still get the desired result?
  - … changes in e.g. CLI APIs?
  - … host-specific configurations?
  - … shell interoperability?
  - … different platforms?

"But my task is dead-simple …

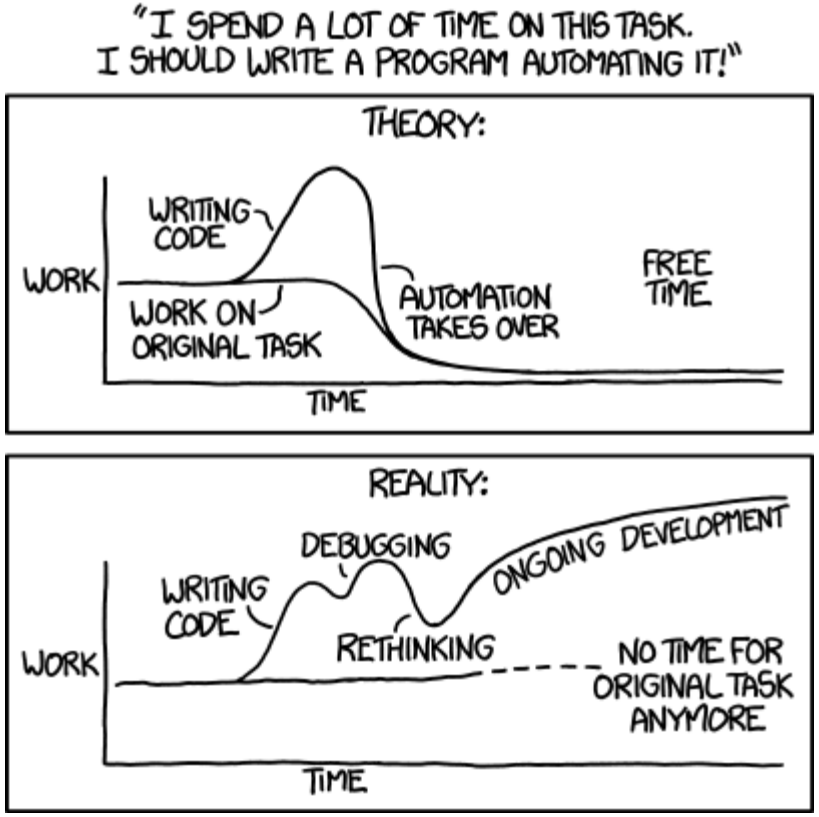
- How does your shell script handle …
  - … if you need to run it only partially?
  - … if it needs to be understood by humans?
  - … if it needs to be run sequentially on multiple hosts?
  - … if it needs to be run on multiple hosts in parallel?
  - …
- → simple tasks usually turn out harder than expected

we're talking about Ansible here, but other configuration management/IaC tools would get the job done as well – a matter of requirements and taste

Open Questions

not specific to Ansible

- When to automate?
  - When expected to do task more than two times?
  - Depends on complexity
- Programming vs. markup
  - When to write a, for example, Python program instead of using Ansible?



https://xkcd.com/1319/

Overview

- Intro
- Cloud Operating Systems
  - OpenStack
- Infrastructure-as-Code
  - Ansible
- **Container Orchestration**
  - Kubernetes
- DevOps, Continuous Integration, and Continuous Delivery
  - Spinnaker, GitLab

Container Analogy

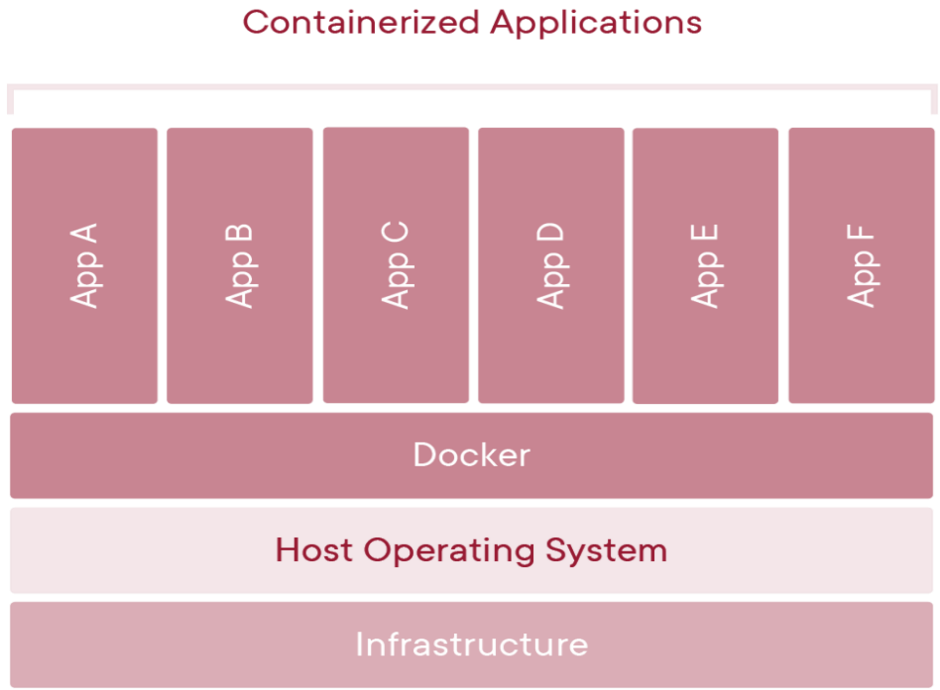
- Containers
  - Standardized
  - Easy to move
  - Isolated
  - Many containers fit hosts

- Dependency management: libs and config bundled with the application to run it everywhere → build your app in a container, test the container locally or your staging environment, then ship the container

Recap: Docker

- Container instance: running an isolated process, started from an image
- Docker image: snapshot of a container, packaging the application, its dependencies, and configuration
- Docker file: instructions for creating images

- Multiple containers share single OS kernel, isolated by kernel containment features (e.g. cgroups)



Dockerfiles

- Creating docker images from configuration files
- Example: Dockerfile for an Express web server image using NodeJS

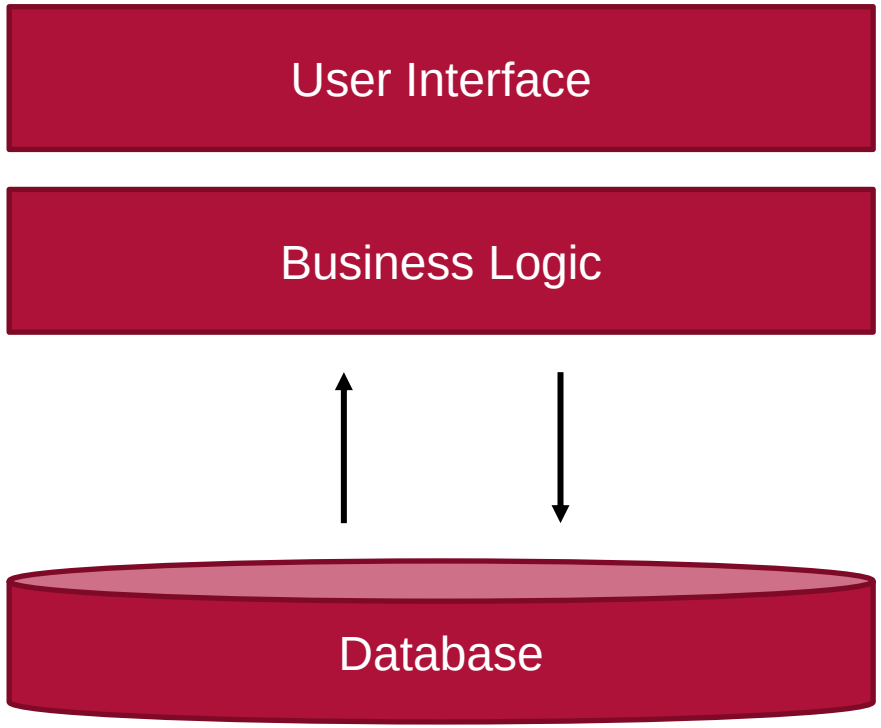
```
$ docker build --tag hello .

$ docker run --rm hello
```

```
# Create image based on the official Node 6 image from dockerhub
FROM node:9

# Create a dir where our app will be placed
RUN mkdir -p /usr/src/app

# Change dir so that our commands run inside this new dir
WORKDIR /usr/src/app

# Copy dependency definitions
COPY package.json /usr/src/app

# Install dependecies
RUN npm install

# Get all the code needed to run the app
COPY . /usr/src/app

# Expose the port the app runs in
EXPOSE 3000

# Serve the app
CMD ["npm", "start"]
```

Container Composition

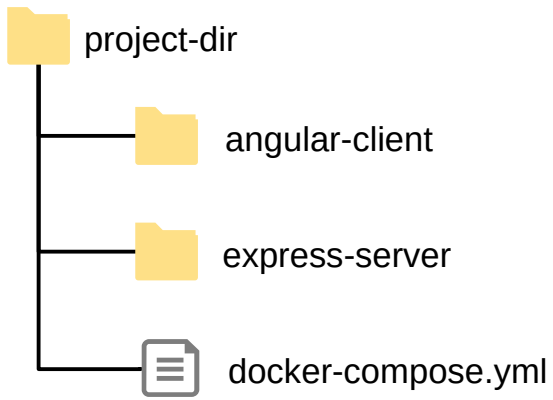
- Applications typically consist of multiple connected components intended to run as a single service

- e.g. Web Application
  - Front-end framework
  - Back-end framework
  - Persistent data store



Docker Compose

- Tool for defining and running multi-container Docker applications on one node

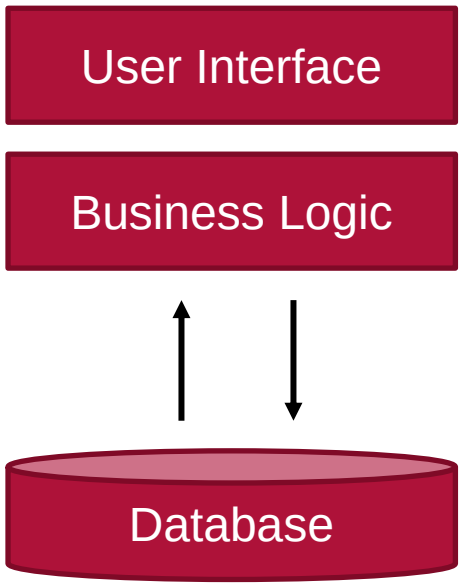



```
$ docker-compose up
```

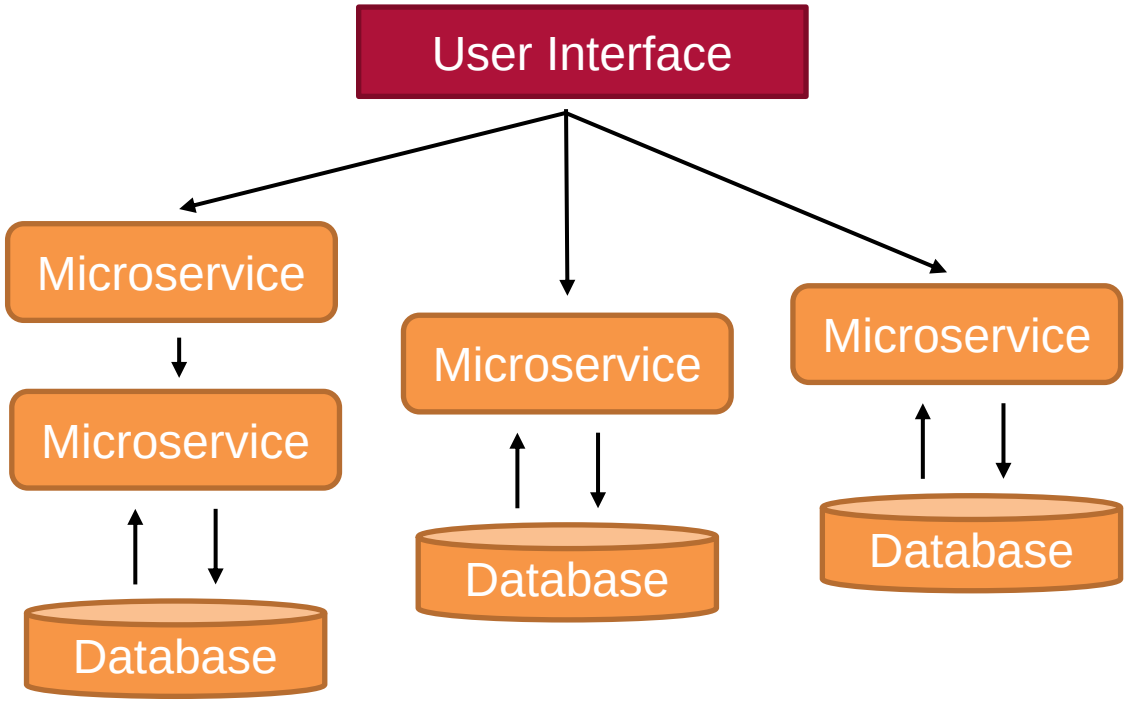
```
version: '2' # specify docker-compose version
# Define the services/containers to be run
services:
  angular:
    build: angular-client # Dockerfile directory
    ports:
      - "4200:4200" # port forewarding

  express:
    build: express-server # Dockerfile directory
    ports:
      - "3000:3000" # port forewarding

  database:
    image: mongo # image to build container from
    ports:
      - "27017:27017" # port forewarding
```

Microservices (1/2)

Monolith

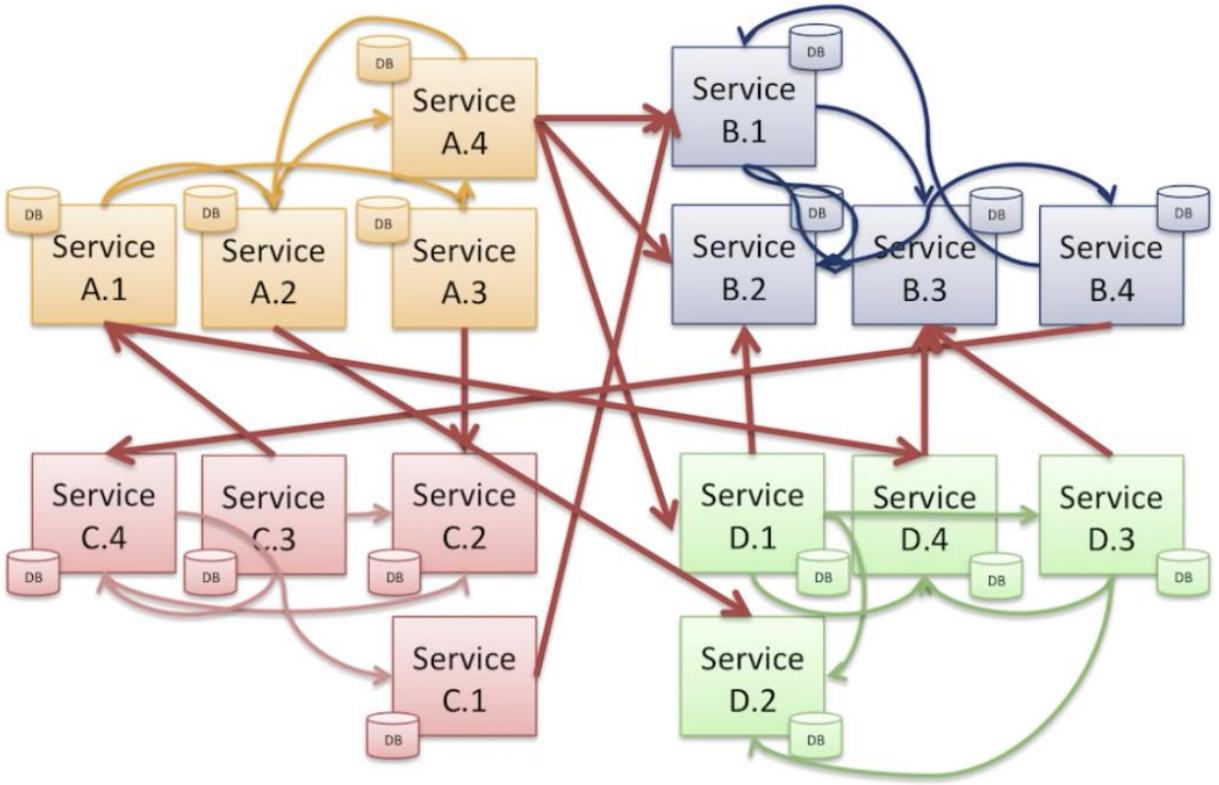


Microservices



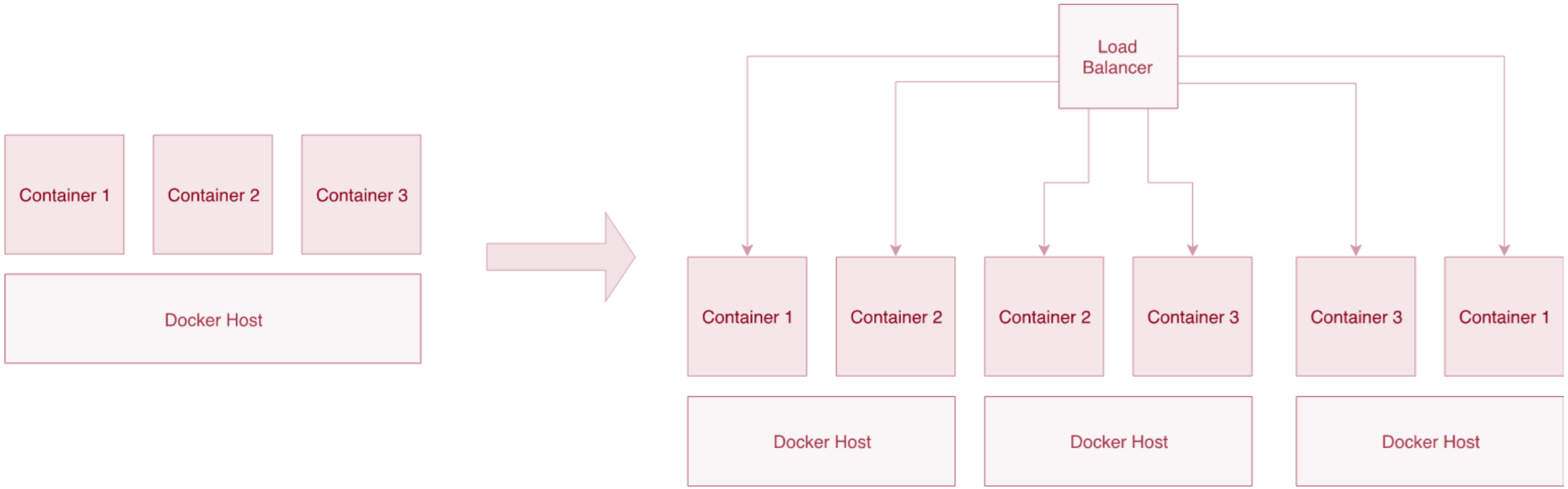
Microservices (2/2)

- Advantages
  - Independent development
  - Small teams
  - Fault isolation
  - Scalable
- Disadvantages
  - Overhead (duplicated tech)
  - Management of services and networking

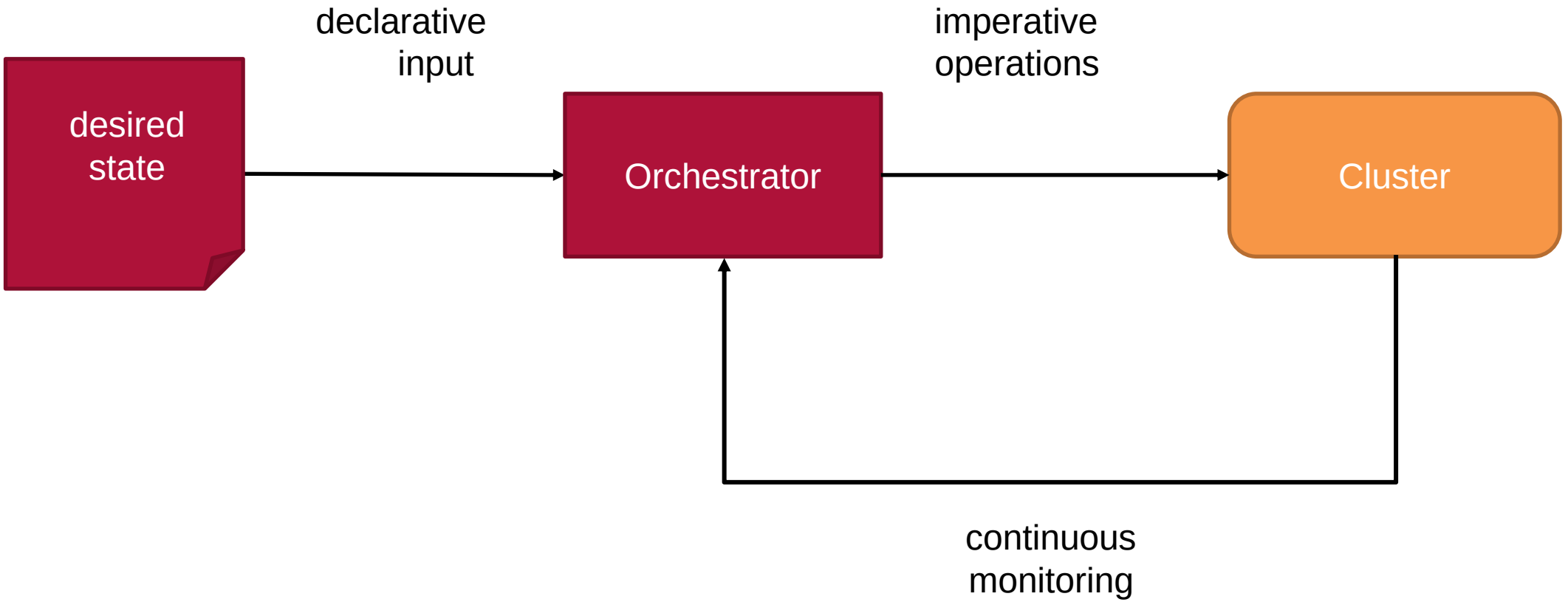


Docker Cluster

- Running an application on a cluster of docker hosts for fault tolerance and scalability



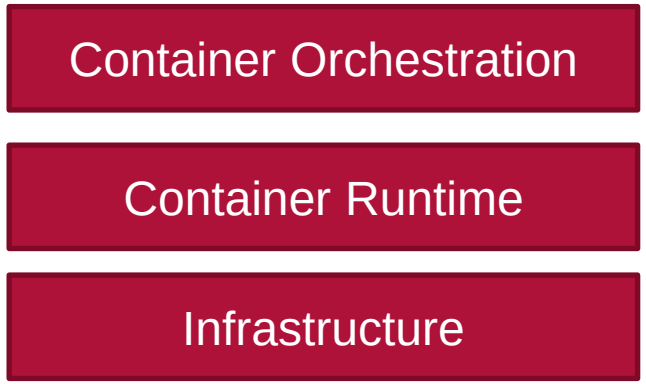
Orchestration: Control systems for clusters



Main Orchestration Tasks

- Scheduling and placement
- Service configuration
- Networking and storage management
- Monitoring and logging
- Replication and scaling
- Re-scheduling and load balancing
- Rolling updates

Container Orchestration



Distributed container management

Local container management

Container-agnostic infrastructure

- Provisioning and deployment of containers
- Configuration and networking of containers
- Replication and availability of containers
- Monitoring of replicated containers
- Load balancing across replicated containers

Overview

- Intro
- Cloud Operating Systems
  - OpenStack
- Infrastructure-as-Code
  - Ansible
- Container Orchestration
  - **Kubernetes**
- DevOps, Continuous Integration, and Continuous Delivery
  - Spinnaker, GitLab

Kubernetes

- Greek for “helmsman” or “pilot”, abbreviated ”k8s”

- Open-sourced by Google in 2014
- Version 1.0 released in July 2015

- Google and the Linux Foundation created the Cloud Native Computing Foundation (CNCF) to offer k8s as an open standard

- Google Kubernetes Engine (GKE), Azure Container Service (AKS), AWS Elastic Container Service (EKS)

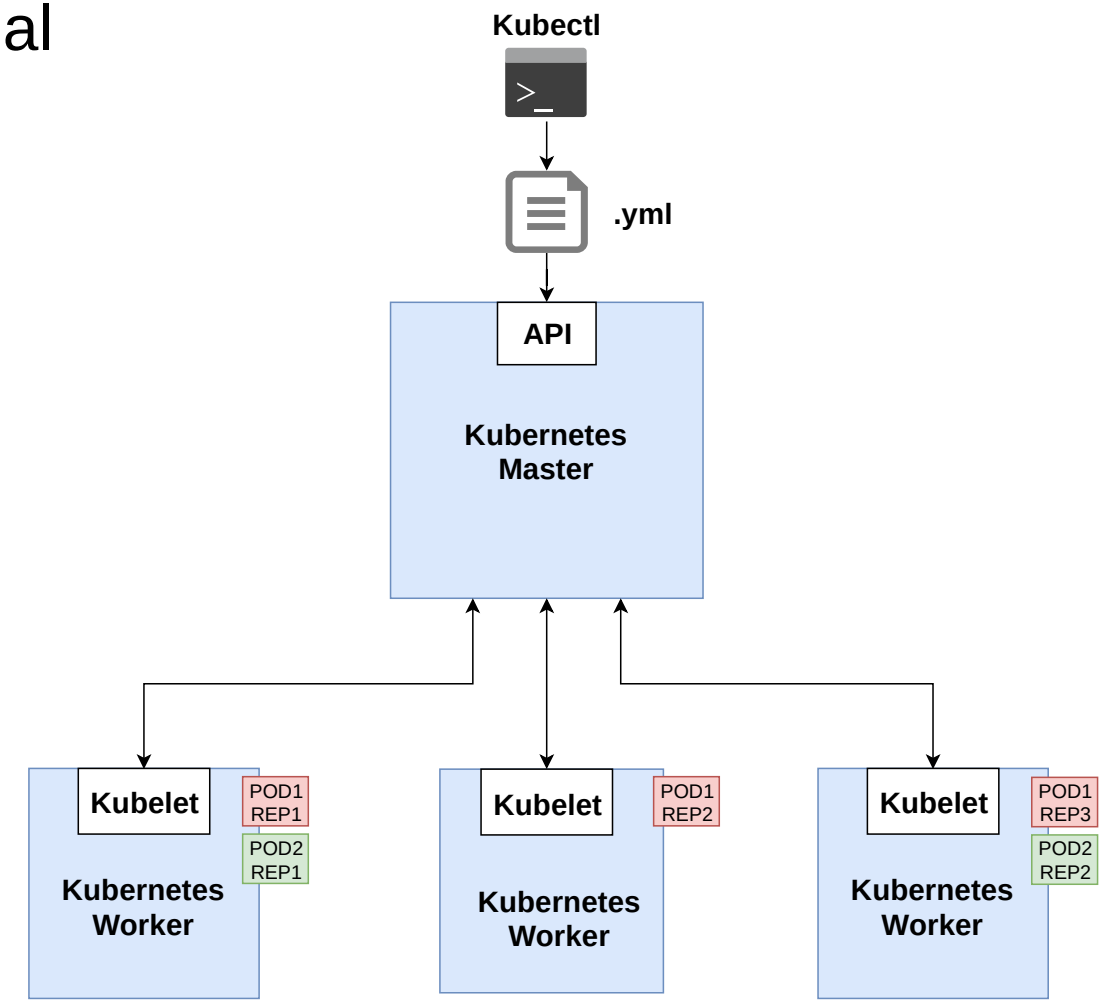
What Does k8s Do for You?

- Automation engine for cluster management, based on a declarative description of the desired cluster state

- Deployment, monitoring, recovery, and scaling
  - Instantiating sets of containers
  - Linking containers together through agreed interfaces
  - Exposing services to machines outside the cluster
  - Monitoring, logging, and re-scheduling of failed containers
  - Dynamically scaling the cluster by adding or removing containers

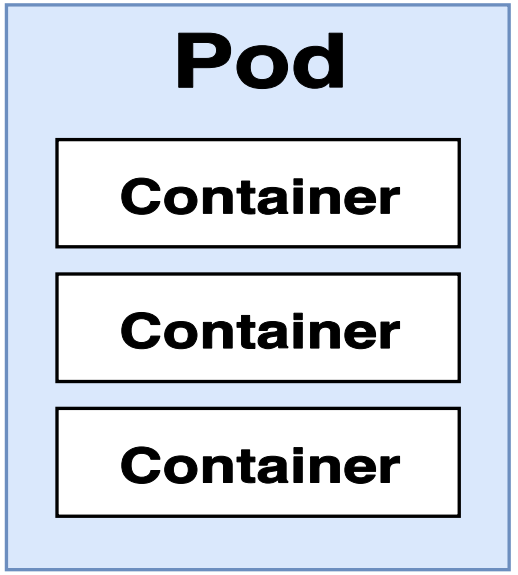
Kubernetes Cluster

- Collection of nodes (either bare-metal or virtual machines), managed by Kubernetes
- Runs groups of replicated containers (which are called Pods)



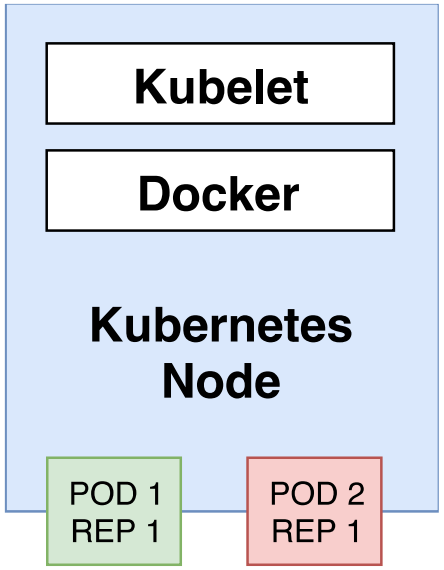
Kubernetes Pods

- Pods consist of
  - Group of containers
  - Container configurations
  - Shared storage
- Containers in a pod
  - are scheduled together
  - are guaranteed to be on the same node
- Replicas of pods



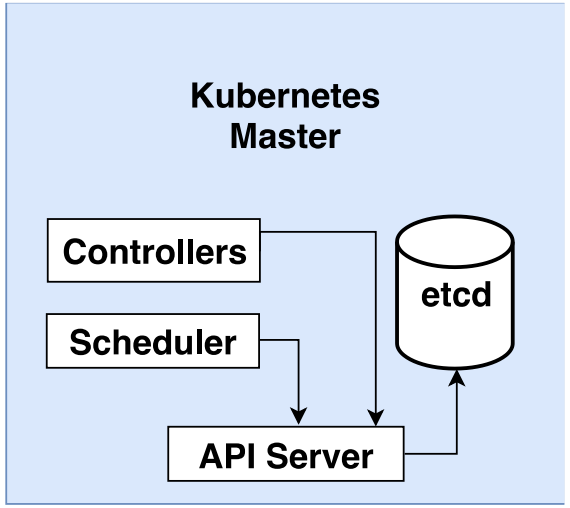
Kubernetes Nodes

- Each worker node in the cluster runs two processes:
  - `kubelet`: agent that communicates withthe Kubernetes master (master → cluster)
  - `kube-proxy`: makes defined services available on each node (cluster → master)



Kubernetes Master

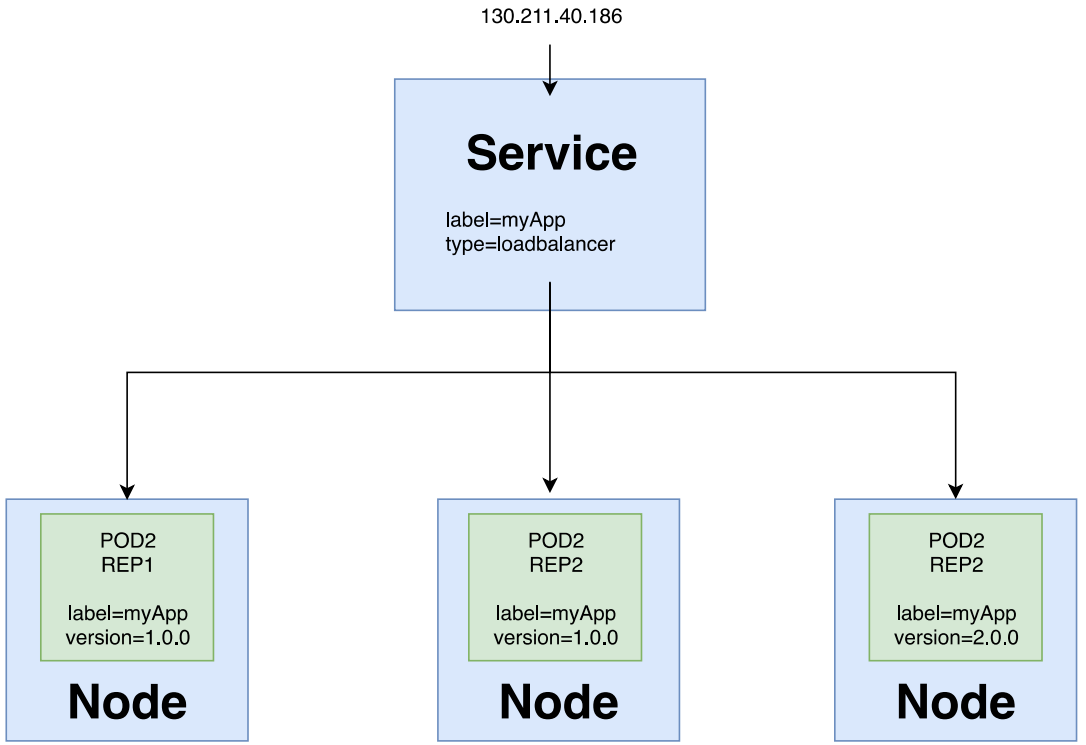
- Collection of processes managing the server state on a single node of the cluster
- Controllers, e.g. replication controller
- Scheduler: places pods based on resource requirements, hardware, and software constraints, data locality, deadlines…
- etcd: reliable distributed key-value store for cluster state



Kubernetes Services

- Services expose functionality of pods in the cluster and to the outside
- Example: load balancer service

```
apiVersion: v1
kind: Service
metadata:
  name: myApp-service
spec:
  type: LoadBalancer
  ports:
  - protocol: TCP
    # accessible externally
    nodePort: 80
    # port in Pod
    targetPort: 8080
  selector:
    app: myApp
```



Kubernetes Cluster State

- Kubernetes allows to specify the state of pods in a cluster and then manages the cluster accordingly
  - *Deployments* for stateless pods:
    - Do not keep a record of pod state and do not need permanent storage and IDs for networking
    - Used e.g. for web server pods (like Nginx and Apache)
  - *StatefulSets* for stateful pods:
    - Keep track of pod state by saving it to a storage and pods communicate using persistent unique IDs
    - Used e.g. for database pods (like MongoDB)

Kubernetes Deployment

- Specifies the deployment of a stateless pod, which k8s then establishes (and re-establishes)

```
$ kubectl create -f nginx-deployment.yaml
```

```
$ kubectl get deployments
```

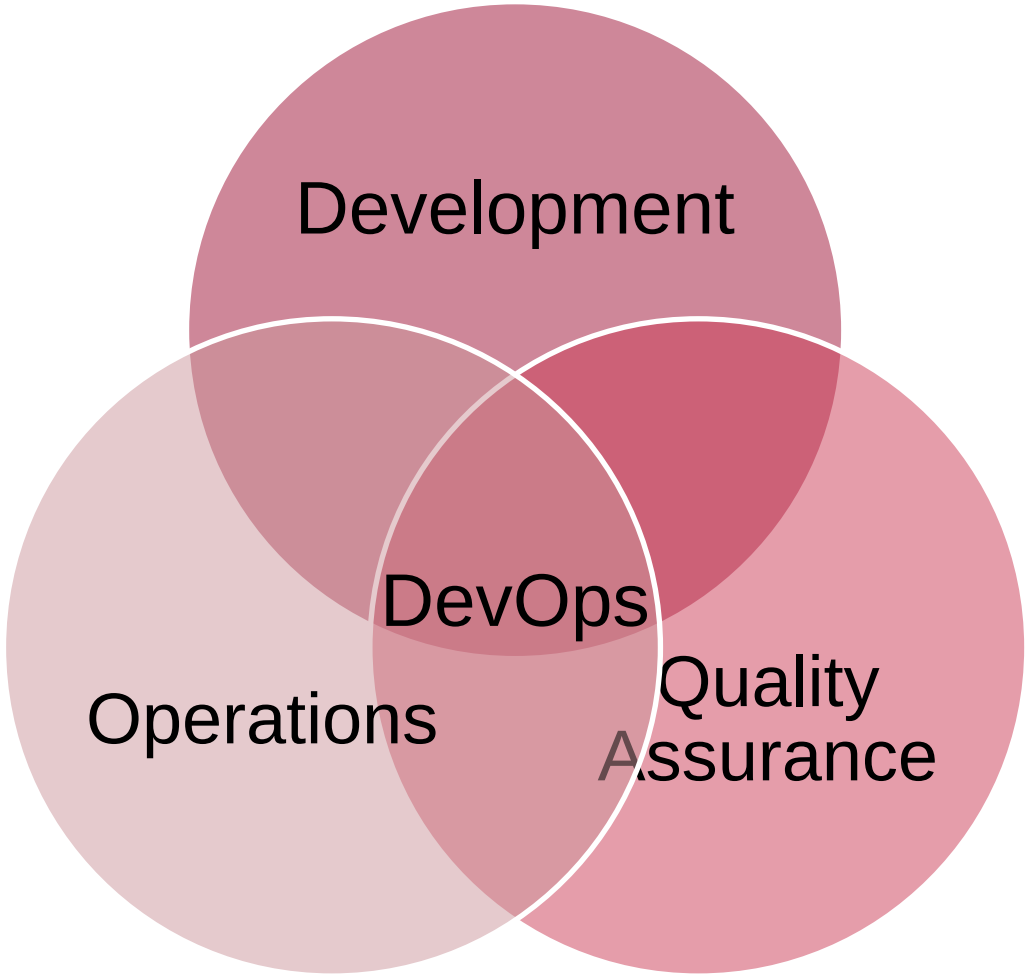
```
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           18s
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels: app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.4
        ports:
        - containerPort: 80
```

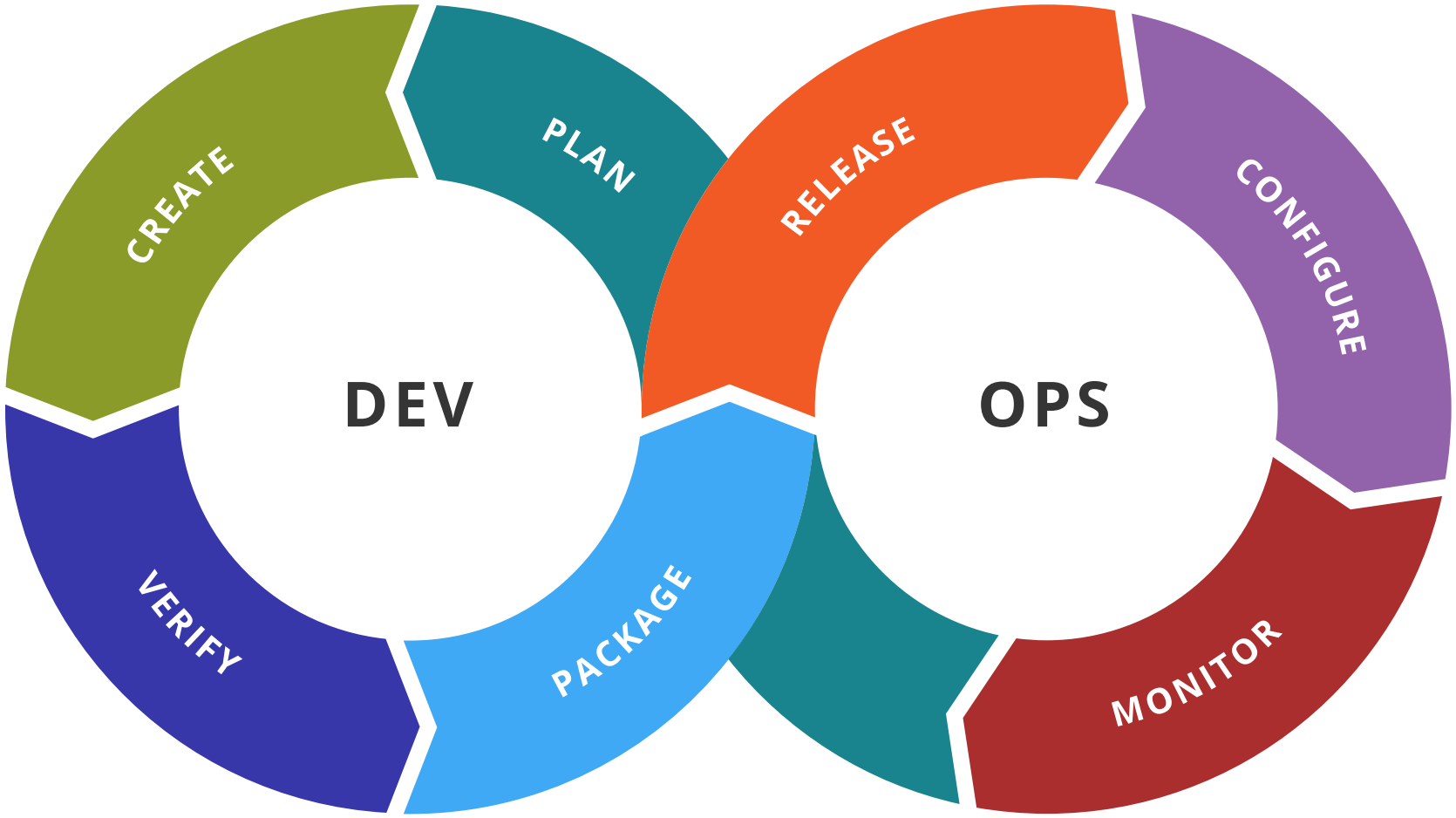
Overview

- Intro
- Cloud Operating Systems
  - OpenStack
- Infrastructure-as-Code
  - Ansible
- Container Orchestration
  - Kubernetes
- **DevOps, Continuous Integration, and Continuous Delivery**
  - Spinnaker, GitLab

DevOps



DevOps


CREATE
PLAN
RELEASE
CONFIGURE
DEV
OPS
VERIFY
PACKAGE
MONITOR

Continuous Integration

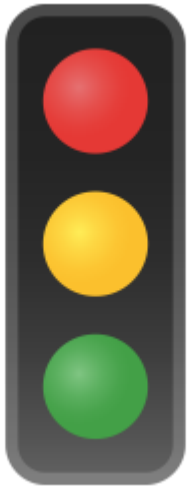
- Automation of *integration and testing* of software

- First proposed by Grady Booch, 1991
- Adopted and popularized as part of XP

- Motivation:
  - Evade “integration hell”: Find bugs early, while still easy to fix
  - Immediate feedback on system-wide impact of local changes
  - Team has a common view of the state of a software project

Continuous Integration: Practices

- One common (versioned) code repository
- Build automation (and short builds)
- Self-testing builds
- Regular commits
- Building every commit (usually on a CI server)
- Test & production environments as similar as possible
- Test results visible for everyone



Continuous Delivery

- Fast and reproducible software releases

- From the first principle of the agile manifesto:

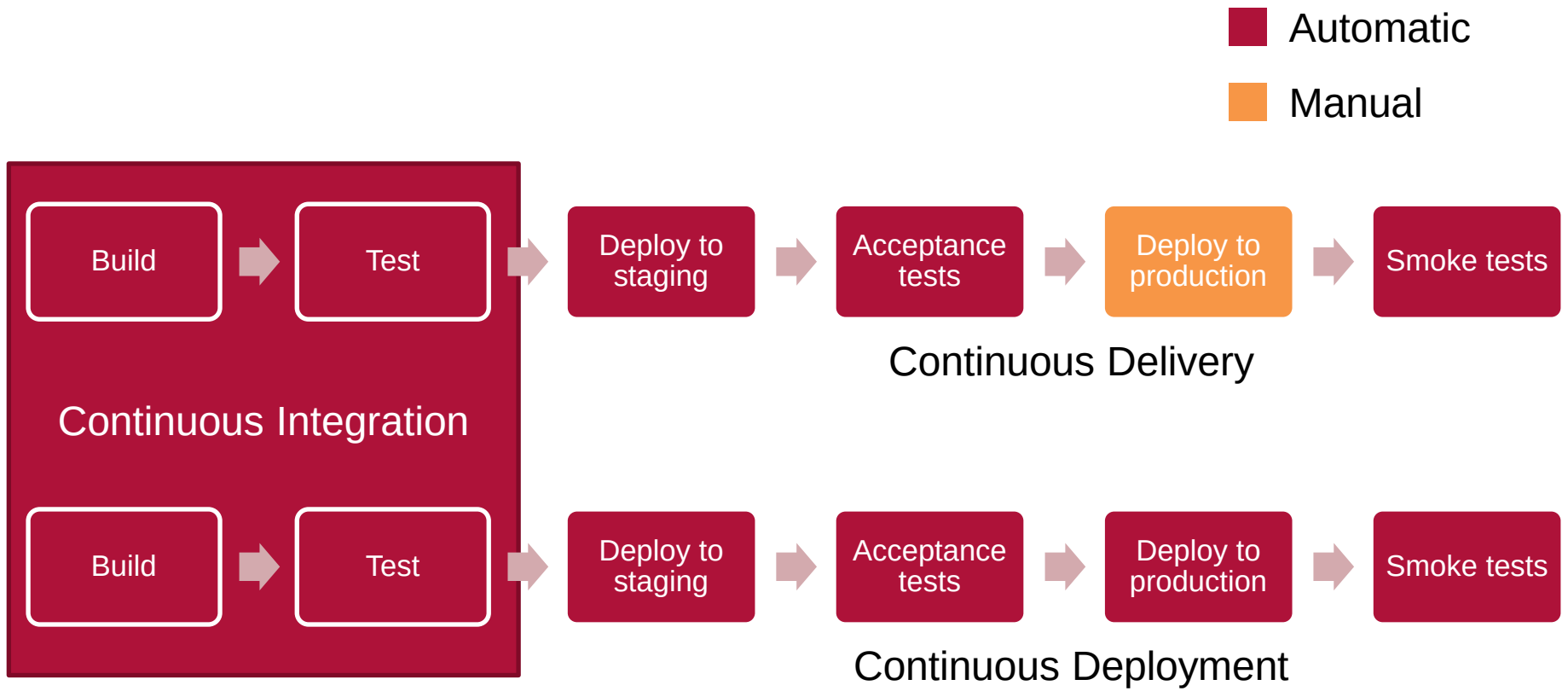
  “Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.”

- CD Metric: How long does it take to release a single-line-code-change to production? *(cycle time)*
- Business case: Software development as investment → optimizing ROI through small cycle time

Continuous Delivery: Problems of Delivering Software

- Antipatterns
  - Deploying software manually
  - Deploying to a production-like environment only after development is complete
  - Manual configuration management of production environments

Continuous Delivery vs. Continuous Deployment



Continuous Delivery: Principles

- Parts of a working software application:
  - Executable code
  - Configuration
  - Host environment
  - Data
- Feedback on changes to any part
- Feedback as fast as possible: unit tests, acceptance tests, capacity tests

Deployment Strategies: Blue/Green

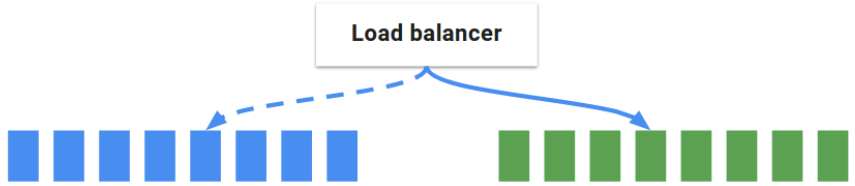
- Strategy to deploy an upgrade
- Two production environments: Blue and green
- Only one environments is active at a time

1. Deploy to the non-active environment
2. Switch the traffic coming through the load balancer over to the new environment
3. Monitor new version, maybe switch back

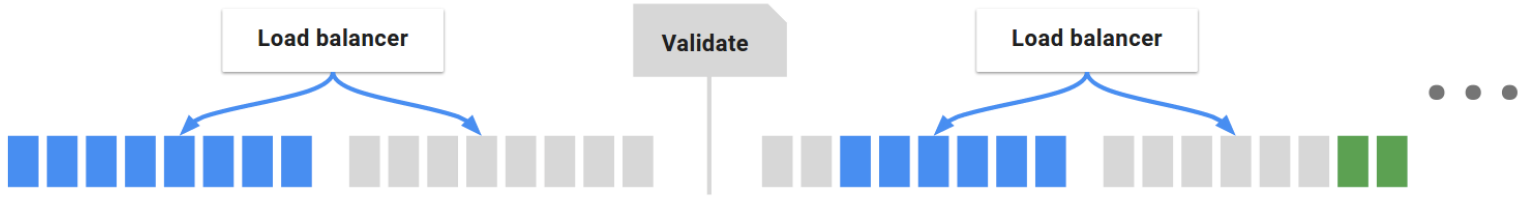
- One environment always runs the last version, the other the newest version: easy rollbacks

Deployment Strategies

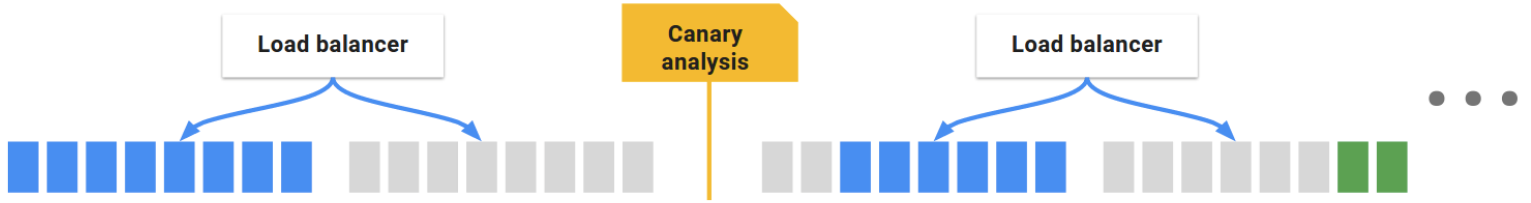
Red/black
(Blue/green)



Rolling red/black



Canary



Overview

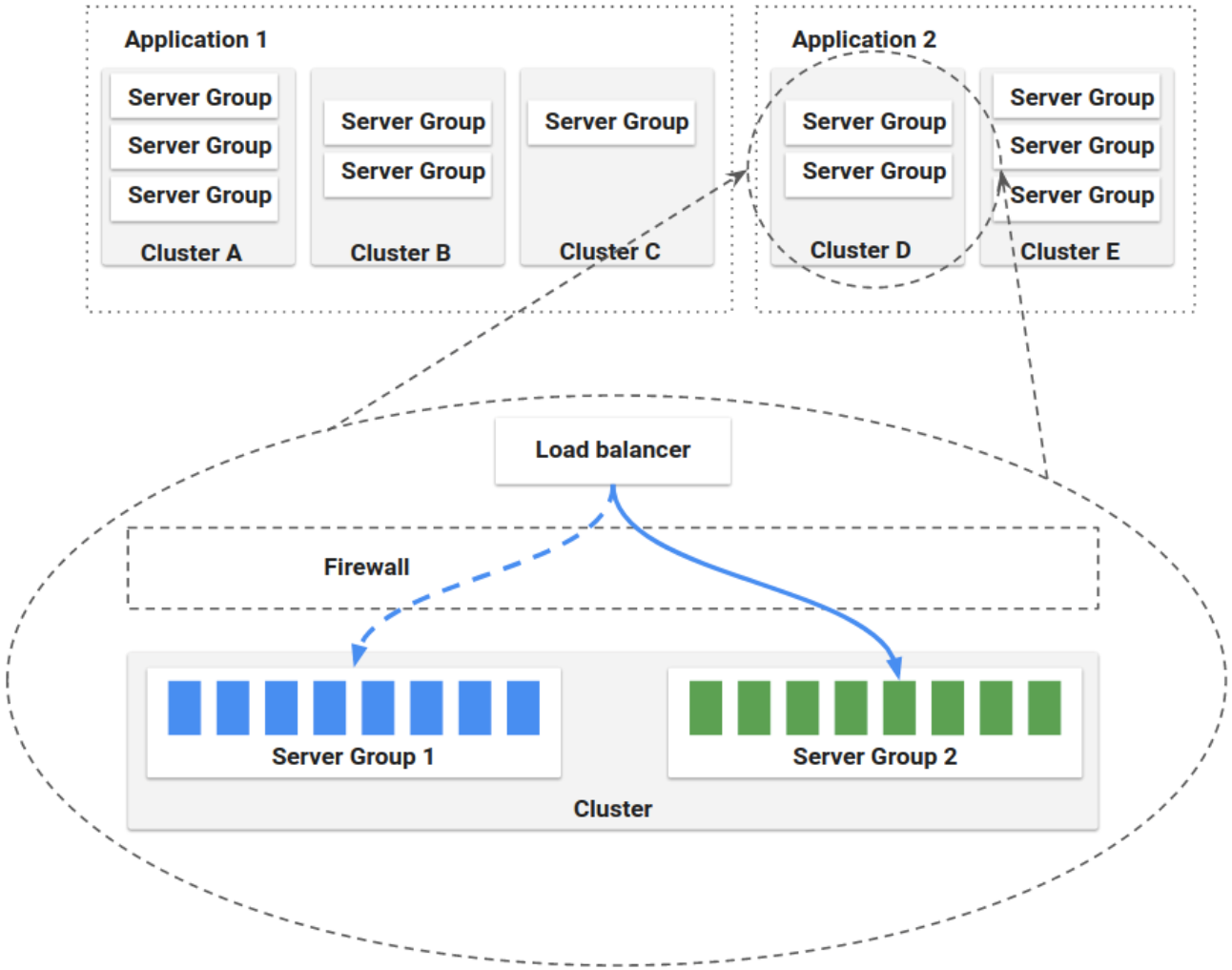
- Intro
- Cloud Operating Systems
  - OpenStack
- Infrastructure-as-Code
  - Ansible
- Container Orchestration
  - Kubernetes
- DevOps, Continuous Integration, and Continuous Delivery
  - **Spinnaker, GitLab**

Spinnaker

- Continuous delivery platform
- Started by Netflix, picked up and extended by Google
  - Initial release November 2015
  - Successor to Asgard, a web interface for automated AWS deployments
  - From baking to cloud deployment
- Support and common abstractions for EC2, GCE, Azure, Kubernetes...

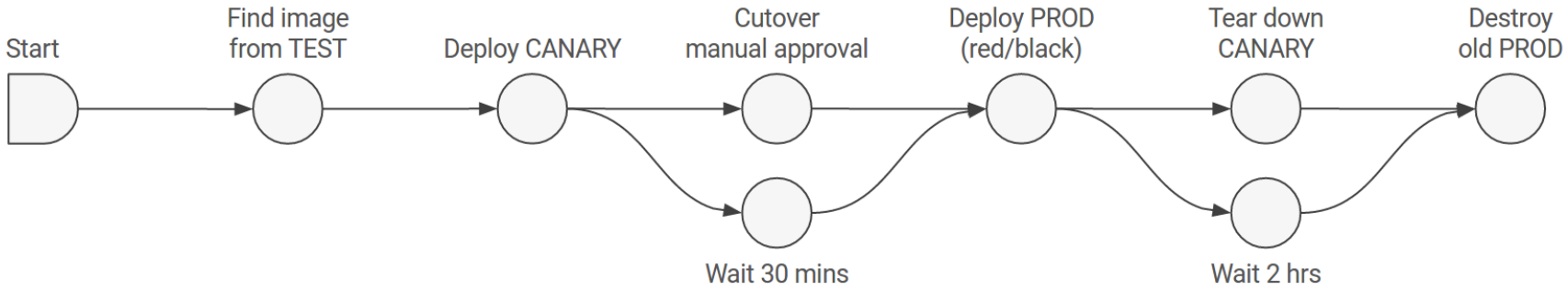
Spinnaker

- Deploy applications / (micro)services
- Organize servers into
  - Clusters
  - Server groups

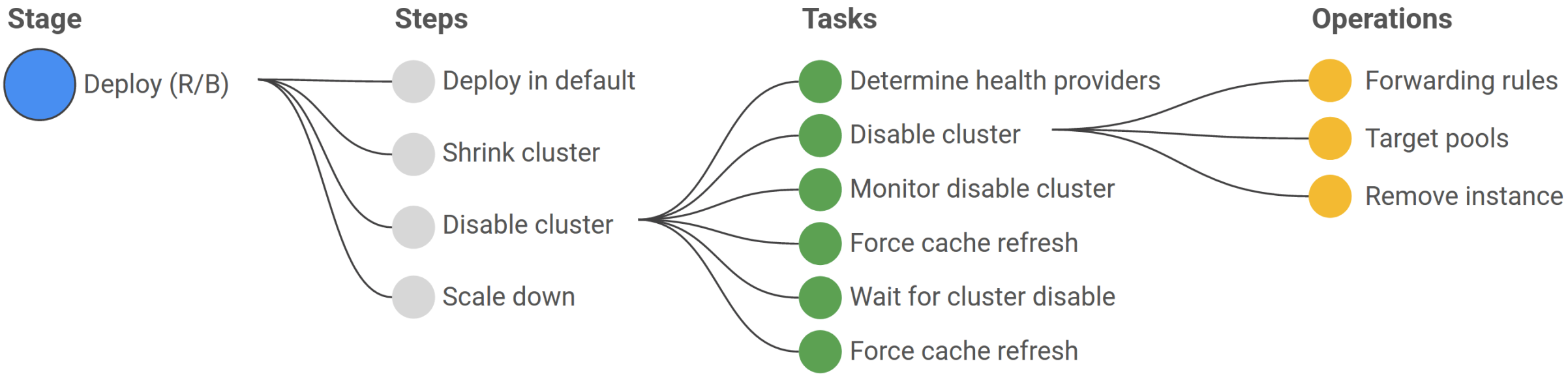


Spinnaker Pipeline

- Deployment happens via pipelines
- Pipelines are built from stages
- Predefined stages exist for many things
  - Bake
  - Manual judgement
  - Clone server group
  - Deploy



Spinnaker Stage Example



GitLab CI/CD

- GitLab Includes a CI/CD system
- CI Jobs are defined in .gitlab-ci.yml

Job Name

```
#gitlab-ci.yml

image: blang/latex
build:
    script:
        - latexmk -pdf
    only:
        - master
    artifacts:
        paths:
            - "*.pdf"
```

Commands for this job (a required property)

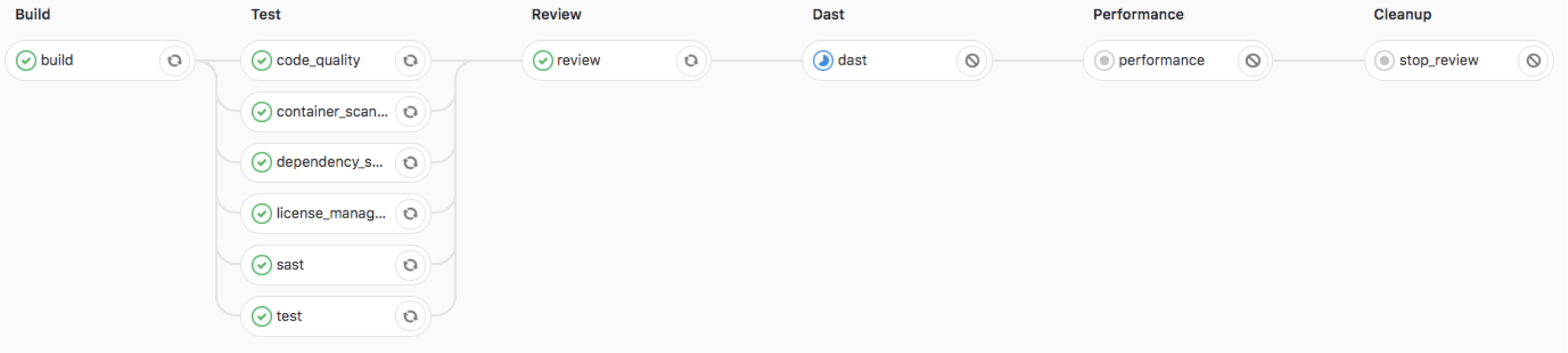
Only execute on certain branches

Save these artifacts back to server

GitLab: Auto DevOps

- Based on Docker, Kubernetes and Gliderlabs Herokuish
- Defined as one long .gitlab-ci.yml
- Builds based on
  - Dockerfile or
  - Herokuish buildpacks
- Tests
  - Code Quality
  - Dependencies
  - Static and dynamic application security testing
  - Test via buildpacks
- Deploy to production or new review app

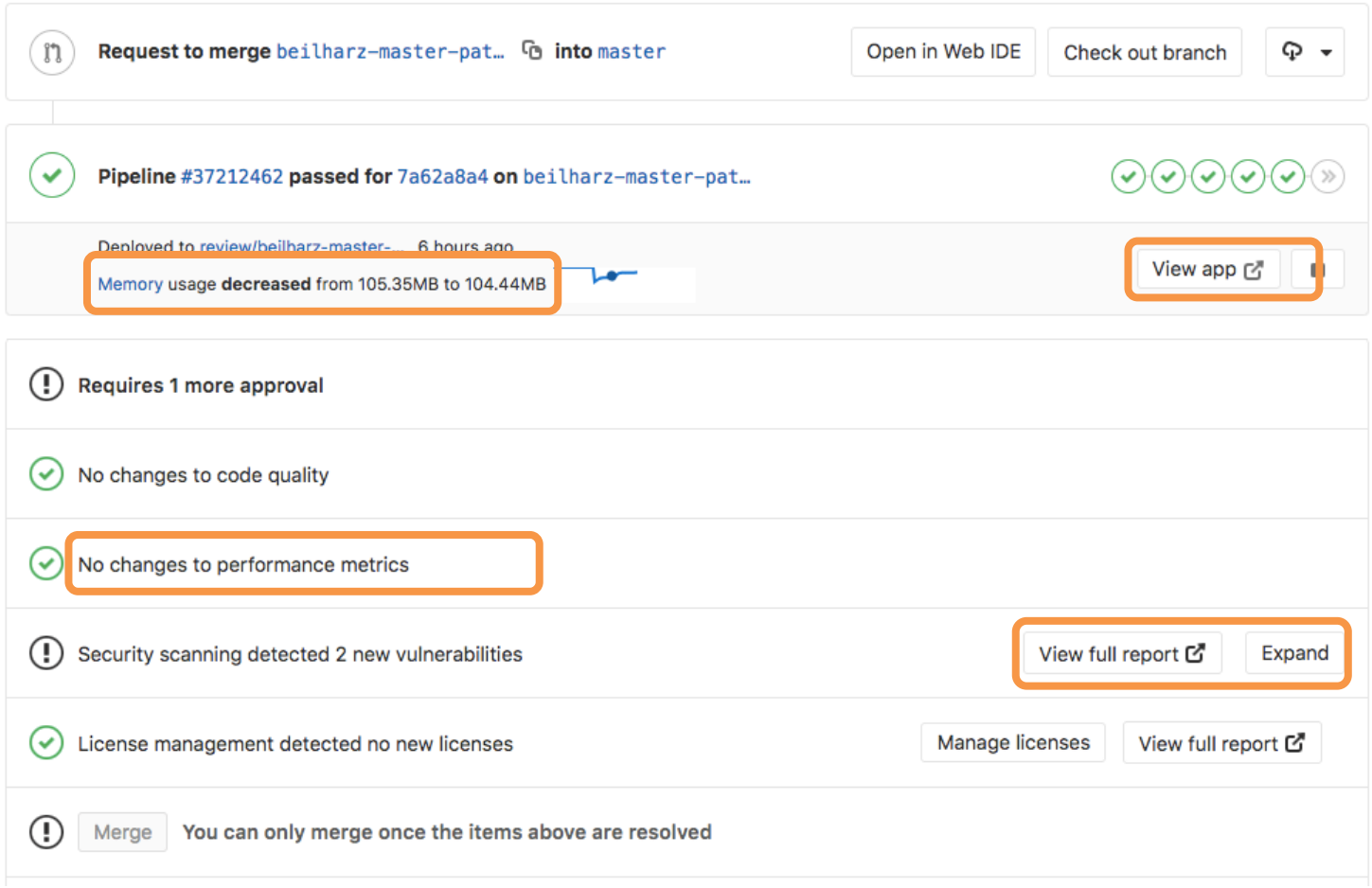
GitLab: Auto DevOps, Example



GitLab pipeline for the branches of a project

GitLab: Auto DevOps

- Integrated with the merge request flow



Overview

- Intro
- Cloud Operating Systems
  - OpenStack
- Infrastructure-as-Code
  - Ansible
- Container Orchestration
  - Kubernetes
- DevOps, Continuous Integration, and Continuous Delivery
  - Spinnaker, GitLab

Summary

- Virtual machines and containers allow to quickly provision new servers, i.e. by using the interface of a Cloud OS
- Yet, server management and configuration remain very challenging (server sprawl, configuration drift, snowflake servers)
- Central Ideas
  - Infrastructure-as-Code: executable, understandable, and versioned descriptions of desired states as input to automation and orchestration tools
  - Cultural change to DevOps, for continuously and automatically deploying new versions to production

References

- Humble, Jez, and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Pearson Education, 2010.
- https://martinfowler.com/bliki/SnowflakeServer.html
- https://martinfowler.com/bliki/BlueGreenDeployment.html
- https://www.spinnaker.io/concepts/
- https://www.spinnaker.io/reference/architecture/
- https://docs.gitlab.com/ee/ci/
- https://docs.gitlab.com/ee/topics/autodevops/index.html